MQSeries®

Application Messaging Interface

SC34-5604-00

Note!

Before using this information and the product it supports, be sure to read the general information under Appendix C, “Notices” on page 321.

First edition (September 1999)

This edition applies to IBM® MQSeries® Application Messaging Interface Version 1, and to any subsequent releases and modifications until otherwise indicated in new editions.

Contents

Figures

Tables

Tables

About this book

This book describes how to use the MQSeries Application Messaging Interface. The Application Messaging Interface provides a simple interface that application programmers can use without needing to understand all the details of the MQSeries Message Queue Interface.

Format of this book

This book is available in portable document format (PDF) only. To view it you need the Adobe Acrobat Reader, Version 3 or later. Click on an entry in the table of contents, or a cross reference within the text, to move directly to that page. Use the Acrobat Reader controls to return to the previous page.

This book is not available in hard copy.

Who this book is for

This book is for anyone who wants to use the Application Messaging Interface to send and receive MQSeries messages, including publish/subscribe and point-to-point applications.

What you need to know to understand this book

- Knowledge of the C, C++, or Java™ programming language is assumed.
- You don't need previous experience of MQSeries to use the Application Messaging Interface (AMI). You can use the examples and sample programs provided to find out how to send and receive messages. However, in order to understand all the functions of the AMI you need to have some knowledge of the MQSeries Message Queue Interface (MQI). This is described in the *MQSeries Application Programming Guide* and the *MQSeries Application Programming Reference* book.
- If you are a systems administrator responsible for setting up an installation of the AMI, you need to be experienced in using the MQI.

Structure of this book

This book contains the following parts:

- Part 1, "Introduction" on page 1 gives an overview of the Application Messaging Interface.
- Part 2, "The C interface" on page 9 describes how to use the AMI in C programs. If you are new to MQSeries, gain some experience with the C high-level interface first. It provides most of the functionality you need when writing applications. Then move on to the C object interface if you need extra functionality.
- Part 3, "The C++ interface" on page 117 describes how to use the AMI in C++ programs.
- Part 4, "The Java interface" on page 191 describes how to use the AMI in Java programs.

- Part 5, “Setting up an AMI installation” on page 257 is for systems administrators who are setting up an Application Messaging Interface installation.

Appearance of text in this book

This book uses the following type styles:

`Format`	The name of a parameter in an MQSeries call, a field in an MQSeries structure, or an attribute of an MQSeries object
dltmqbrk	The name of an AMI function or method
AMB_TRUE	The name of an AMI constant
`amInitialize`	The syntax of AMI functions and methods, and example code

MQSeries publications

This section describes MQSeries publications that are referred to in this manual. They are available in hardcopy, HTML and PDF formats, except where noted.

MQSeries Application Programming Guide

The *MQSeries Application Programming Guide*, SC33-0807, provides guidance information for users of the message queue interface (MQI). It describes how to design, write, and build an MQSeries application. It also includes full descriptions of the sample programs supplied with MQSeries.

MQSeries Application Programming Reference

The *MQSeries Application Programming Reference*, SC33-1673, provides comprehensive reference information for users of the MQI. It includes: data-type descriptions; MQI call syntax; attributes of MQSeries objects; return codes; constants; and code-page conversion tables.

MQSeries Publish/Subscribe User’s Guide

The *MQSeries Publish/Subscribe User’s Guide*, GC34-5269, provides comprehensive information for users of the MQSeries Publish/Subscribe SupportPac™. It includes: installation; system design; writing applications; and managing the publish/subscribe broker.

This book is available in PDF format only.

MQSeries System Administration

The *MQSeries System Administration* book, SC33-1873, supports day-to-day management of local and remote MQSeries objects. It includes topics such as security, recovery and restart, transactional support, problem determination, and the dead-letter queue handler. It also includes the syntax of the MQSeries control commands.

MQSeries information on the Internet

MQSeries Web site

The MQSeries product family Web site is at:

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

By following links from this Web site you can:

- Obtain latest information about the MQSeries product family.
- Access the MQSeries books in HTML and PDF formats.
- Download MQSeries SupportPacs.

Portable Document Format (PDF)

PDF files can be viewed and printed using the Adobe Acrobat Reader. It is recommended that you use Version 3 or later.

If you need to obtain the Adobe Acrobat Reader, or would like up-to-date information about the platforms on which the Acrobat Reader is supported, visit the Adobe Systems Inc. Web site at:

```
http://www.adobe.com/
```

MQSeries on the Internet

Part 1. Introduction

Chapter 1. Introduction

The MQSeries products enable programs to communicate with one another across a network of dissimilar components - processors, operating systems, subsystems, and communication protocols - using a consistent application programming interface, the MQSeries *Message Queue Interface* (MQI). The purpose of the *Application Messaging Interface* (AMI) is to provide a simple interface that application programmers can use without needing to understand all the functions available in the MQI. The functions that are required in a particular installation are defined by a system administrator, using *services* and *policies*.

Main features of the AMI

There are three main components in the AMI:

- The message, which defines *what* is sent from one program to another
- The service, which defines *where* the message is sent
- The policy, which defines *how* the message is sent

To send a message using the AMI, an application has to specify the message data together with the service and policy to be used. You can use the default services and policies provided by the system, or create your own. Optionally, you can store your definitions of services and policies in a *repository*.

Sending and receiving messages

You can use the AMI to send and receive messages in a number of different ways:

- Send and forget (datagram), where no reply is needed
- Distribution list, where a message is sent to multiple destinations
- Request/response, where a sending application needs a response to the request message
- Publish/subscribe, where a broker manages the distribution of messages

Interoperability

The AMI is interoperable with other MQSeries interfaces. Using the AMI you can exchange messages with one or more of the following:

- Another application that is using the AMI
- Any application that is using the MQI
- A message broker (such as MQSeries Publish/Subscribe or MQSeries Integrator)

Programming languages

The Application Messaging Interface is available in the C, C++ and Java programming languages. In C there are two interfaces: a high-level interface that is procedural in style, and a lower level object-style interface. The high-level interface contains the functionality needed by the majority of applications. The two interfaces can be mixed as required.

In C++ and Java, a single object interface is provided.

Description of the AMI

In the Application Messaging Interface, messages, services and policies define what is being sent, where it is sent, and how it is sent.

Messages

Information is passed between communicating applications using messages, with MQSeries providing the transport. Messages consist of:

- The message attributes: information that identifies the message and its properties. The AMI uses the attributes, together with information in the policy, to interpret and construct MQSeries headers and message descriptors.
- The message data: the application data carried in the message. The AMI does not act upon this data.

Some examples of message attributes are:

MessageID	An identifier for the message. It is usually unique, and typically it is generated by the message transport (MQSeries).
CorrelID	A correlation identifier that can be used as a key, for example to correlate a response message to a request message. The AMI normally sets this in a response message by copying the *MessageID* from the request message.
Format	The structure of the message.
Topic	Indicates the content of the message for publish/subscribe applications.

These attributes are properties of an AMI message object. Where it is appropriate, an application can set them before sending a message, or access them after receiving a message. The message data can be contained in the message object, or passed as a separate parameter.

In an MQSeries application, the message attributes are set up explicitly using the Message Queue Interface (MQI), so the application programmer needs to understand their purpose. With the AMI, they are contained in the message object or defined in a policy that is set up by the system administrator, so the programmer is not concerned with these details.

Services

A service represents a destination that applications send messages to or receive messages from. In MQSeries such a destination is called a *message queue*, and a queue resides in a *queue manager*. Programs can use the MQI to put messages on queues, and get messages from them. Because there are many parameters associated with queues and the way they are set up and managed, this interface is complex. When using the AMI, these parameters are defined in a service that is set up by the systems administrator, so the complexity is hidden from the application programmer.

For further information about queues and queue managers, please refer to the *MQSeries Application Programming Guide*.

Point-to-point and publish/subscribe

In a *point-to-point* application, the sending application knows the destination of the message. Point-to-point applications can be send and forget (or datagram), where a reply to the message is not required, or request/response, where the request message specifies the destination for the response message. Applications using distribution lists to send a message to multiple destinations are usually of the send and forget type.

In the case of *publish/subscribe* applications, the providers of information are decoupled from the consumers of that information. The provider of the information is called a *publisher*. Publishers supply information about a subject by sending it to a broker. The subject is identified by a *topic*, such as "Stock" or "Weather". A publisher can publish information on more than one topic, and many publishers can publish information on a particular topic.

The consumer of the information is called a *subscriber*. A subscriber decides what information it is interested in, and subscribes to the relevant topics by sending a message to the broker. When information is published on one of those topics, the publish/subscribe broker sends it to the subscriber (and any others who have registered an interest in that topic). Each subscriber is sent information about those topics it has subscribed to.

There can be many brokers in a publish/subscribe system, and they communicate with each other to exchange subscription requests and publications. A publication is propagated to another broker if a subscription to that topic exists on the other broker. So a subscriber that subscribes to one broker will receive publications (on a chosen topic) that are published at another broker.

The AMI provides functions to send and receive messages using the publish/subscribe model. For further details see the *MQSeries Publish/Subscribe User's Guide*.

Types of service

Different types of service are defined to specify the mapping from the AMI to real resources in the messaging network.

- Senders and receivers establish one-way communication pipes for sending and receiving messages.
- A distribution list contains a list of senders to which messages can be sent.

- A publisher contains a sender that is used to publish messages to a publish/subscribe broker.
- A subscriber contains a sender, used to subscribe to a publish/subscribe broker, and a receiver, for receiving publications from the broker.

The AMI provides default services that are used unless otherwise specified by the application program. You can define your own service when calling a function, or use a customized service stored in a *repository* (these are set up by a systems administrator). You don't have to have a repository. Many of the options used by the services are contained in a policy (see below).

The AMI has functions to open and close services explicitly, but they can also be opened and closed implicitly by other functions.

Policies

A policy controls how the AMI functions operate. The AMI provides default policies. Alternatively, a systems administrator can define customized policies and store them in a repository. An application program selects a policy by specifying it as a parameter on calls.

Policies control such items as:

- The attributes of the message, for example the priority.
- Options used for send and receive operations, for instance whether it is part of a unit of work.
- Publish/subscribe options, for example whether a publications is retained.
- Added value functions to be invoked as part of the call, such as retry.

You could choose to use a different policy on each call, and specify in the policy only those parameters that are relevant to the particular call. You could then have policies shared between applications, such as a "Transactional_Persistent_Put" policy. Another approach is to have policies that specify all the parameters for all the calls made in a particular application, such as a "Payroll_Client" policy. Both approaches are valid with the AMI, but a single policy for each application will simplify management of policies.

The AMI will automatically retry when temporary errors are encountered on sending a message, if requested by the policy. (Examples of temporary errors are queue full, queue disabled, and queue in use).

Application Messaging Interface model

Figure 1 shows the components of the Application Messaging Interface.

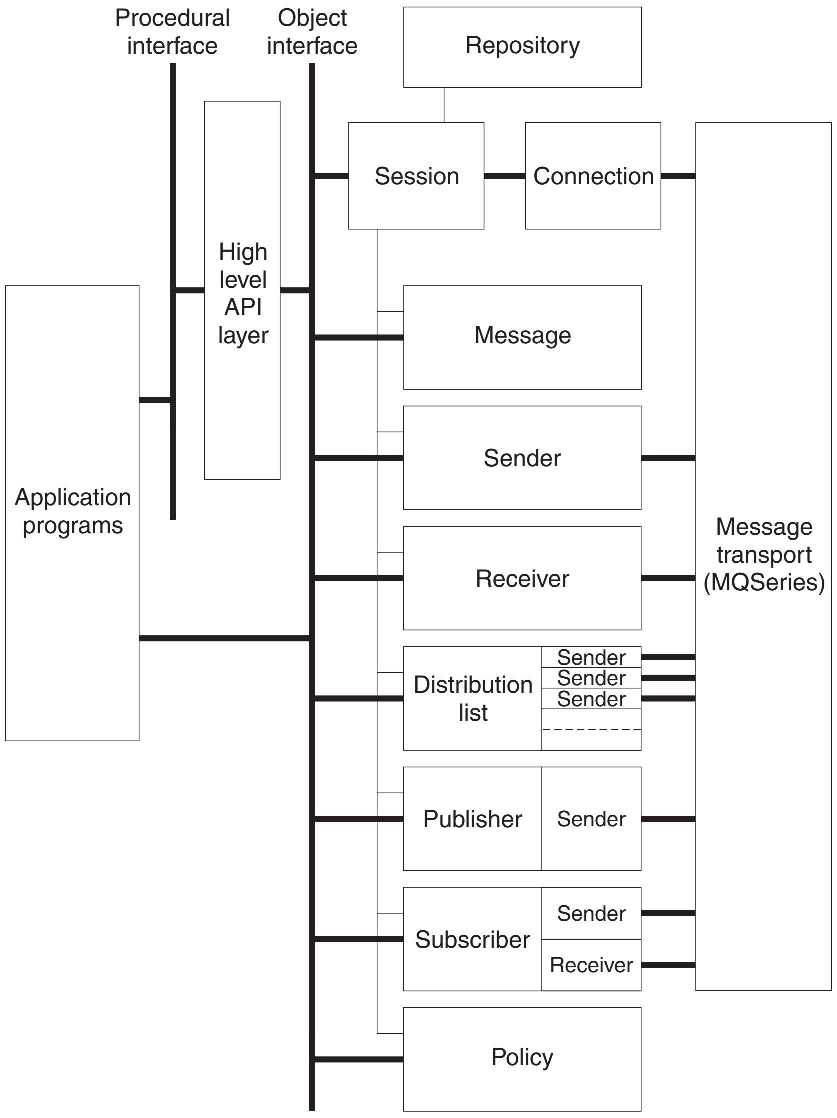

Figure 1. Basic AMI model

Application programs communicate directly with AMI objects using the object interface in C, C++ and Java. In addition to the C object-style interface, there is a procedural-style high-level interface available in C. This contains the functionality needed by the majority of applications; it can be supplemented with object interface functions as needed.

Sender, receiver, distribution list, publisher, and subscriber objects are all services. Senders and receivers connect directly to the message transport layer (MQSeries). Distribution list and publisher objects contain senders; subscriber objects contain a sender and a receiver.

Message, service and policy objects are created and managed by a session object, which provides the scope for a unit of work. The session object contains a connection object that is not visible to the application. The combination of connection, sender, and receiver objects provides the transport for the message. Other objects, such as helper classes, are provided in C++ and Java.

Attributes for message, service and policy objects can be taken from the system defaults, or from administrator-provided definitions that have been stored in the repository.

Further information

The Application Messaging Interface is available for the C, C++, and Java programming languages. Although the concepts are the same, the syntax differs according to the language, so the implementation for each language is described in a separate part of this book:

- Part 2, “The C interface” on page 9
- Part 3, “The C++ interface” on page 117
- Part 4, “The Java interface” on page 191

In Part 5, “Setting up an AMI installation” on page 257, you can find out how to:

- Install the Application Messaging Interface
- Run the sample programs
- Determine the cause of problems
- Set up services and policies

The Application Messaging Interface runs on the following operating systems or environments: AIX®, Sun Solaris, Microsoft® Windows® 98 and Windows NT®.

Part 2. The C interface

This part contains:

Chapter 2. Using the Application Messaging Interface in C

The Application Messaging Interface (AMI) in the C programming language has two interfaces:

1. A high-level procedural interface that provides the function needed by the majority of users.
2. A lower-level, object-style interface, that provides additional function for experienced MQSeries users.

This chapter describes the following:

- "Structure of the AMI"
- "Writing applications in C" on page 14
- "Building C applications" on page 24

Structure of the AMI

Although the high-level interface is procedural in style, the underlying structure of the AMI is object based. (The term *object* is used here in the object-oriented programming sense, not in the sense of MQSeries 'objects' such as channels and queues.) The objects that are made available to the application are:

Session	Contains the AMI session.
Message	Contains the message data, message ID, correlation ID, and options that are used when sending or receiving a message (most of which come from the policy definition).
Sender	This is a service that represents a destination (such as an MQSeries queue) to which messages are sent.
Receiver	This is a service that represents a source from which messages are received.
Distribution list	Contains a list of sender services to provide a list of destinations.
Publisher	Contains a sender service where the destination is a publish/subscribe broker.
Subscriber	Contains a sender service (to send subscribe and unsubscribe messages to a publish/subscribe broker) and a receiver service (to receive publications from the broker).
Policy	Defines how the message should be handled, including items such as priority, persistence, and whether it is included in a unit of work.

When using the high-level functions the objects are created automatically and (where applicable) populated with values from the repository. In some cases it might be necessary to inspect these properties after a message has been sent (for instance, the `MessageID`), or to change the value of one or more properties before sending the message (for instance, the `Format`). To satisfy these requirements, the AMI for C has a lower-level object style interface in addition to the high-level procedural interface. This provides access to the objects listed above, with methods to *set* and *get* their properties. You can mix high-level and object-level functions in the same application.

All the objects have both a *handle* and a *name*. The names are used to access objects from the high-level interface. The handles are used to access them from the object interface. Multiple objects of the same type can be created with the same name, but are usable only from the object interface.

The high-level interface is described in Chapter 3, “The C high-level interface” on page 29. An overview of the object interface is given in Chapter 4, “C object interface overview” on page 47, with reference information in Chapter 5, “C object interface reference” on page 59.

Using the repository

You can run AMI applications with or without a repository. If you don’t have a repository, you can use a system default object (see below), or create your own by specifying its name on a function call. It will be created using the appropriate system provided definition (see “System provided definitions” on page 280).

If you have a repository, and you specify the name of an object on a function call that matches a name in the repository, the object will be created using the repository definition. (If no matching name is found in the repository, the system provided definition will be used.)

System default objects

Table 1. System default objects

Default object	Constant or handle (if applicable)
SYSTEM.DEFAULT.POLICY	AMSD_POL AMSD_POL_HANDLE
SYSTEM.DEFAULT.SYNCPOINT.POLICY	AMSD_SYNC_POINT_POL AMSD_SYNC_POINT_POL_HANDLE
SYSTEM.DEFAULT.SENDER	AMSD_SND
SYSTEM.DEFAULT.RESPONSE.SENDER	AMSD_RSP_SND AMSD_RSP_SND_HANDLE
SYSTEM.DEFAULT.RECEIVER	AMSD_RCV AMSD_RCV_HANDLE
SYSTEM.DEFAULT.PUBLISHER	AMSD_PUB AMSD_PUB_SND
SYSTEM.DEFAULT.SUBSCRIBER	AMSD_SUB AMSD_SUB_SND
SYSTEM.DEFAULT.SEND.MESSAGE	AMSD_SND_MSG AMSD_SND_MSG_HANDLE
SYSTEM.DEFAULT.RECEIVE.MESSAGE	AMSD_RCV_MSG AMSD_RCV_MSG_HANDLE

A set of system default objects is created at session creation time. This removes the overhead of creating the objects from applications using these defaults. The system default objects are available for use from both the high-level and object interfaces in C. They are created using the system provided definitions (see “System provided definitions” on page 280).

The default objects can be specified explicitly using AMI constants, or used to provide defaults if a parameter is omitted (by specifying NULL, for example).

Constants representing synonyms for handles are also provided for these objects, for use from the object interface (see Appendix B, “Constants” on page 313). Note that the first parameter on a call must be a real handle; you cannot use a synonym in this case (that is why handles are not provided for all the default objects).

Writing applications in C

This section gives a number of examples showing how to use the high-level interface of the AMI, with some extensions using the object interface. Equivalent operations to all high-level functions can be performed using combinations of object interface functions (see "High-level functions" on page 58).

Opening and closing a session

Before using the AMI, you must open a session. This can be done with the following high-level function (page 34):

Opening a session

```
hSession = amInitialize(name, myPolicy, &compCode, &reason);
```

The `name` is optional, and can be specified as NULL. `myPolicy` is the name of the policy to be used during initialization of the AMI. You can specify the policy name as NULL, in which case the SYSTEM.DEFAULT.POLICY object is used.

The function returns a *session handle*, which must be used by other calls in this session. Errors are returned using a completion code and reason code.

To close a session, you can use this high-level function (page 44):

Closing a session

```
success = amTerminate(&hSession, myPolicy, &compCode, &reason);
```

This closes and deletes all objects that were created in the session. Note that a *pointer* to the session handle is passed. If the function is successful, it returns AMB_TRUE.

Sending messages

You can send a datagram (send and forget) message using the high-level **amSendMsg** function (page 40). In the simplest case, all you need to specify is the session handle returned by **amInitialize**, the message data, and the message length. Other parameters are set to NULL, so the default message, sender service, and policy objects are used.

Sending a message using all the defaults

```
success = amSendMsg(hSession, NULL, NULL, dataLen,
       pData, NULL, &compCode, &reason);
```

If you want to send the message using a different sender service, specify its name (such as `mySender`) as follows:

Sending a message using a specified sender service

```
success = amSendMsg(hSession, mySender, NULL, dataLen,
       pData, NULL, &compCode, &reason);
```

If you are not using the default policy, you can specify a policy name:

Sending a message using a specified policy

```
success = amSendMsg(hSession, NULL, myPolicy, dataLen,
       pData, NULL, &compCode, &reason);
```

The policy controls the behavior of the send function. For example, the policy can specify:

- The priority, persistence and expiry of the message
- If the send is part of a unit of work
- If sender service should be implicitly opened and left open

To send a message to a distribution list, specify its name (such as `myDistList`) as the sender service:

Sending a message to a distribution list

```
success = amSendMsg(hSession, myDistList, NULL, dataLen,
       pData, NULL, &compCode, &reason);
```

Using the message object

Using the object interface gives you more functions when sending a message. For example, you can *get* or *set* individual attributes in the message object. To get an attribute after the message has been sent, you can specify a name for the message object that is being sent:

Specifying a message object

```
success = amSendMsg(hSession, NULL, NULL, dataLen,
       pData, mySendMsg, &compCode, &reason);
```

The AMI creates a message object of the name specified (`mySendMsg`), if one doesn't already exist. (The sender name and policy name are specified as NULL, so in this example their defaults are used.) You can then use object interface functions to get the required attributes, such as the *MessageID*, from the message object:

Getting an attribute from a message object

```
hMsg = amSesGetMessageHandle(hSession, mySendMsg, &compCode, &reason);

success = amMsgGetMsgId(hMsg, BUFLEN, &MsgIdLen, pMsgId,
       &compCode, &reason);
```

The first call is needed to get the handle to the message object. The second call returns the message ID length, and the message ID itself (in a buffer of length BUFLEN).

To set an attribute such as the *Format* before the message is sent, you must first create a message object and set the format:

Setting an attribute in a message object

```
hMsg = amSesCreateMessage(hSession, mySendMsg, &compCode, &reason);

success = amMsgSetFormat(hMsg, AMLEN_NULL_TERM, pFormat,
            &compCode, &reason);
```

Then you can send the message as before, making sure to specify the same message object name (`mySendMsg`) in the **amSendMsg** call.

Look at "Message interface functions" on page 50 to find out what other attributes of the message object you can get and set.

After a message object has been used to send a message, it might not be left in the same state as it was prior to the send. Therefore, if you use the message object for repeated send operations, it is advisable to reset it to its initial state (see **amMsgReset** on page 82) and rebuild it each time.

Instead of sending the message data using the data buffer, it can be added to the message object. However, this is not recommended for large messages because of the overhead of copying the data into the message object before it is sent (and also extracting the data from the message object when it is received).

Sample programs

For more details, refer to the `amtshsnd.c` and `amtsosnd.c` sample programs (see "The sample programs" on page 277).

Receiving messages

Use the **amReceiveMsg** high-level function (page 36) to receive a message to which no response is to be sent (such as a datagram). In the simplest case, all you need to specify are the session handle and a buffer for the message data. Other parameters are set to NULL, so the default message, receiver service, and policy objects are used.

Receiving a message using all the defaults

```
success = amReceiveMsg(hSession, NULL, NULL, NULL, BUFLEN,
       &dataLen, pData, NULL, &compCode, &reason);
```

If you want to receive the message using a different receiver service, specify its name (such as `myReceiver`) as follows:

Receiving a message using a specified receiver service

```
success = amReceiveMsg(hSession, myReceiver, NULL, NULL, BUFLEN,
       &dataLen, pData, NULL, &compCode, &reason);
```

If you are not using the default policy, you can specify a policy name:

Receiving a message using a specified policy

```
success = amReceiveMsg(hSession, NULL, myPolicy, NULL, BUFLEN,
       &dataLen, pData, NULL, &compCode, &reason);
```

The policy can specify, for example:

- The wait interval
- If the message is part of a unit of work
- If the message should be code page converted
- If all the members of a group must be there before any members can be read

Using the message object

To get the attributes of a message after receiving it, you can specify your own message object name, or use the system default (SYSTEM.DEFAULT.RECEIVE.MESSAGE). If a message object of that name does not exist it will be created. You can access the attributes (such as the *Encoding*) using the object interface functions:

Getting an attribute from a message object

```
success = amReceiveMsg(hSession, NULL, NULL, NULL, BUFLEN,
       &dataLen, pData, myRcvMsg, &compCode, &reason);

hMsg = amSessGetMessageHandle(hSession, myRcvMsg, &compCode, &reason);

success = amMsgGetEncoding(hMsg, &encoding, &compCode, &reason);
```

If a specific message is to be selectively received using its correlation identifier, a message object must first be created and its *CorrelId* property set to the required value (using the object interface). This message object is passed as the *selection message* on the **amReceiverMsg** call:

Using a selection message object

```
hMsg = amSesCreateMessage(hSession, mySelMsg, &compCode, &reason);

success = amMsgSetCorrelId(hMsg, correlIdLen, pCorrelId,
       &compCode, &reason);

success = amReceiveMsg(hSession, NULL, NULL, mySelMsg, BUFLEN,
       &dataLen, pData, NULL, &compCode, &reason);
```

Sample programs

For more details, refer to the `amtshrcv.c` and `amtsorcv.c` sample programs (see “The sample programs” on page 277).

Request/response messaging

In the *request/response* style of messaging, a requester (or client) application sends a request message and expects to receive a message in response. The responder (or server) application receives the request message and produces the response message (or messages) which it returns to the requester application. The responder application uses information in the request message to determine how to send the response message to the requester.

In the following examples ‘your’ refers to the responding application (the server); ‘my’ refers to the requesting application (the client).

Request

Use the **amSendRequest** high-level function (page 41) to send a request message. This is similar to **amSendMsg**, but it includes the name of the service to which the response message is to be sent. In this example the sender service (`mySender`) is specified in addition to the receiver service (`myReceiver`). (A policy name and a send message name can be specified as well, as described in “Sending messages” on page 14).

Sending a request message

```
success = amSendRequest(hSession, mySender, NULL, myReceiver,
       dataLen, pData, NULL, &compCode, &reason);
```

The **amReceiveRequest** high-level function (page 39) is used by the responding (or client) application to receive a request message. It is similar to **amReceiveMsg**, but it includes the name of the sender service that will be used for sending the response message. When the message is received, the sender service is updated with the information needed for sending the response to the required destination.

Receiving a request message

```
success = amReceiveRequest(hSession, yourReceiver, NULL, BUFLEN,
       &dataLen, pData, yourRcvMsg, yourSender,
       &compCode, &reason);
```

A policy name can be specified as well, as described in “Receiving messages” on page 16.

A receiver message name (`yourRcvMsg`) is specified so that the response message can refer to it. Note that, unlike **amReceiveMsg**, this function does not have a selection message.

Response

After the requested actions have been performed, the responding application sends the response message (or messages) with the **amSendResponse** function (page 42):

Sending a response message

```
success = amSendResponse(hSession, yourSender, NULL, yourRcvMsg,
       dataLen, pData, NULL, &compCode, &reason);
```

The sender service for the response message (`yourSender`) and the receiver message name (`yourRcvMsg`) are the same as those used with **amReceiveRequest**. This causes the *CorrelId* and *MessageId* to be set in the response message, as requested by the flags in the request message.

Finally, the requester (or client) application uses the **amReceiveMsg** function to receive the response message as described in “Receiving messages” on page 16. You might need to receive a specific response message (for example if three request messages have been sent, and you want to receive the response to the first request message first). In this case the sender message name from the **amSendRequest** function should be used as the selection message name in the **amReceiveMsg**.

Sample programs

For more details, refer to the `amtshclt.c`, `amtshsvr.c`, `amtsoclt.c`, and `amtsosvr.c` sample programs (see “The sample programs” on page 277).

Publish/subscribe messaging

With *publish/subscribe* messaging, *publisher* applications publish messages to *subscriber* applications using a *broker*. The messages published contain application data and one or more *topic* strings that describe the data. Subscribing applications register subscriptions informing the broker which topics they are interested in. When the broker receives a published message, it forwards the message to all subscribing applications where a topic in the message matches a topic in the subscription.

Publish

Use the **amPublish** high-level function (page 35) to publish a message. You need to specify the name of the publisher for the publish/subscribe broker. The topic relating to this publication and the publication data must also be specified:

Publishing a message

```
success = amPublish(hSession, myPublisher, NULL, myReceiver,
      strlen(topic), pTopic, dataLen, pData, myPubMsg,
      &compCode, &reason);
```

The name `myReceiver` identifies the receiver service to which the broker will send a response message. You can also specify a policy name to change the behavior of the function (as with the **amSend** functions).

You can specify the publication message name `myPubMsg` and set or get attributes of the message object (using the object interface functions). This might include adding another topic (using **amMsgAddTopic**) before invoking **amPublish**, if there are multiple topics associated with this publication.

Instead of sending the publication data using the data buffer, it can be added to the message object. Unlike the **amSend** functions, this gives no difference in performance with large messages. This is because, whichever method is used, the MQRFH header has to be added to the publication data before sending it (similarly the header has to be removed when the publication is received).

Subscribe

The **amSubscribe** high-level function (page 43) is used to subscribe to a publish/subscribe broker specified by the name of a subscriber service. The receiver to which publications will be sent is included within the definition of the subscriber. The name of a receiver service to which the broker can send a response message (`myReceiver`) is also specified.

Subscribing to a broker

```
success = amSubscribe(hSession, mySubscriber, NULL, myReceiver,
       strlen(topic), pTopic, 0L, NULL, mySubMsg,
       &compCode, &reason);
```

A subscription for a single topic can be passed by the `pTopic` parameter. You can subscribe to multiple topics by using the object interface **amMsgAddTopic** function to add topics to the subscription message object, before invoking **amSubscribe**.

If the policy specifies that the *CorrelId* is to be used as part of the identity for the subscribing application, it can be added to the subscription message object with the object interface **amMsgSetCorrelId** function, before invoking **amSubscribe**.

To remove a subscription, use the **amUnsubscribe** high-level function (page 45). To remove all subscriptions, you can specify a policy that has the ‘Deregister All Topics’ subscriber attribute.

To receive a publication from a broker, use the **amReceivePublication** function (page 37). For example:

Receiving a publication

```
success = amReceivePublication(hSession, mySubscriber, NULL, NULL,
       TOPICBUFLEN, BUFLEN, &topicCount, &topicLen, pFirstTopic,
       &dataLen, pData, myRcvMsg, &compCode, &reason);
```

You need to specify the name of the subscriber service used for the original subscription. You can also specify a policy name and a selection message name, as described in “Receiving messages” on page 16, but they are shown as NULL in this example.

If there are multiple topics associated with the publication, only the first one is returned by this function. So, if `topicCount` indicates that there are more topics, you have to access them from the `myRcvMsg` message object, using the object-level **amSesGetMessageHandle** and **amMsgGetTopic** functions.

Sample programs

For more details, refer to the `amtshpub.c`, `amtshsub.c`, `amtsopub.c`, and `amtsosub.c` sample programs (see “The sample programs” on page 277).

Using name/value elements

Publish/subscribe brokers (such as MQSeries Publish/Subscribe) respond to messages that contain name/value pairs to define the commands and options to be used. The **amPublish**, **amSubscribe**, **amUnsubscribe**, and **amReceivePublication** high-level functions provide these name/value pairs implicitly.

For less commonly used commands and options, the name/value pairs can be added to a message using an AMELEM structure, which is defined as follows:

```
 typedef struct tagAMELEM {
   AMCHAR8  strucId;       /* Structure identifier     */
   AMLONG   version;       /* Structure version number */
   AMLONG   groupBuffLen;  /* Reserved, must be zero   */
   AMLONG   groupLen;      /* Reserved, must be zero   */
   AMSTR    pGroup;        /* Reserved, must be NULL   */
   AMLONG   nameBuffLen;   /* Name buffer length       */
   AMLONG   nameLen;       /* Name length in bytes     */
   AMSTR    pName;         /* Name                     */
   AMLONG   valueBuffLen;  /* Value buffer length      */
   AMLONG   valueLen;      /* Value length in bytes    */
   AMSTR    pValue;        /* Value                    */
   AMLONG   typeBuffLen;   /* Reserved, must be zero   */
   AMLONG   typeLen;       /* Reserved, must be zero   */
   AMSTR    pType;         /* Reserved, must be NULL   */
  } AMELEM;
```

As an example, to send a message containing a 'Request Update' command, initialize the AMELEM structure and then set the following values:

Name	`"MQPSCommand"`
Value	`"ReqUpdate"`

Then create a message object (`mySndMsg`) and add the element to it:

Using name/value elements

```
hMsg = amSessCreateMessage(hSession, mySndMsg, &compCode, &reason);

success = amMsgAddElement(hMsg, pElem, 0L, &compCode, &reason);
```

You must then send the message, using **amSendMsg**, to the sender service specified for the publish/subscribe broker.

If you need to use streams with MQSeries Publish/Subscribe, you must add the appropriate stream name/value element explicitly to the message object. Helper macros (such as **AmMsgAddStreamName**) are provided to simplify this and other tasks.

The message element functions can, in fact, be used to add any element to a message before issuing an publish/subscribe request. Such elements (including topics, which are specialized elements) supplement or override those added implicitly by the request, as appropriate to the individual element type.

The use of name/value elements is not restricted to publish/subscribe applications. They can be used in other applications as well.

Error handling

Each AMI C function returns a completion code reflecting the success or failure (OK, warning, or error) of the request. Information indicating the reason for a warning or error is returned in a reason code. Both completion and reason codes are optional.

In addition, each function returns an AMBOOL value or an AMI object handle. For those functions that return an AMBOOL value, this is set to AMB_TRUE if the function completes successfully or with a warning, and AMB_FALSE if an error occurs.

The ‘get last error’ functions (such as **amSesGetLastError**) always reflect the last most severe error detected by an object. These functions can be used to return the completion and reason codes associated with this error. Once the error has been handled, call the ‘clear error codes’ functions (for instance, **amMsgClearErrorCodes**) to clear the error information.

Note that not all C high-level functions record last error information in the session object, but in the underlying named object associated with the error. It can be accessed by obtaining the handle of the underlying object, followed by the relevant ‘get last error’ call (for example, using **amSesGetSenderHandle** followed by **amSndGetLastError**).

Transaction support

Messages sent and received by the AMI can, optionally, be part of a transactional unit of work. A message is included in a unit of work based on the setting of the syncpoint attribute specified in the policy used on the call. The scope of the unit of work is the session handle and only one unit of work may be active at any time.

The API calls used to control the transaction depends on the type of transaction is being used.

- MQSeries messages are the only resource

  A transaction is started by the first message sent or received under syncpoint control, as specified in the policy specified for the send or receive. Multiple messages can be included in the same unit of work. The transaction is committed or backed out using an **amCommit** or **amBackout** high-level interface call (or the **amSesCommit** or **amSesRollback** object-level calls).
- Using MQSeries as an XA transaction coordinator

  The transaction must be started explicitly using the **amSesBegin** call before the first recoverable resource (such as a relational database) is changed. The transaction is committed or backed out using an **amCommit** or **amBackout** high-level interface call (or the **amSesCommit** or **amSesRollback** object-level calls).
- Using an external transaction coordinator

  The transaction is controlled using the API calls of an external transaction coordinator (such as CICS, Encina or Tuxedo). The AMI calls are not used but the syncpoint attributed must still be specified in the policy used on the call.

Other considerations

Multithreading

If you are using multithreading with the AMI, a session normally remains locked for the duration of a single AMI call. If you use receive with wait, the session remains locked for the duration of the wait, which might be unlimited (that is, until the wait time is exceeded or a message arrives on the queue). If you want another thread to run while a thread is waiting for a message, it must use a separate session.

AMI handles and object references can be used on a different thread from that on which they were first created for operations that do not involve an access to the underlying (MQSeries) message transport. Functions such as initialize, terminate, open, close, send, receive, publish, subscribe, unsubscribe, and receive publication will access the underlying transport restricting these to the thread on which the session was first opened (for example, using **amInitialize** or **amSesOpen**). An attempt to issue these on a different thread will cause an error to be returned by MQSeries and a transport error (AMRC_TRANSPORT_ERR) will be reported to the application.

Using MQSeries with the AMI

You must not mix MQSeries function calls with AMI function calls within the same process.

Field limits

When string and binary properties such as queue name, message format, and correlation ID are set, the maximum length values are determined by MQSeries, the underlying message transport. See the rules for naming MQSeries objects in the *MQSeries Application Programming Guide*.

Building C applications

AMI include file

AMI provides an include file, **amtc.h**, to assist you with the writing of your applications. It is recommended that you become familiar with the contents of this file.

The include file is installed under:

```
/amt/inc         (UNIX)

\amt\include     (Windows)
```

See “Directory structure” on page 263 (AIX), 267 (Solaris), or 270 (Windows).

Your AMI C program must contain the statement:

```
#include <amtc.h>
```

The AMI include file must be accessible to your program at compilation time.

Data types

All data types are defined by means of the **typedef** statement. For each data type, the corresponding pointer data type is also defined. The name of the pointer data type is the name of the elementary or structure data type prefixed with the letter "P" to denote a pointer; for example:

```
typedef AMHSES  AMPOINTER PAMHSES;  /* pointer to AMHSES  */
```

Initial values for structures

The include file amtc.h defines a macro variable that provides initial values for the AMELEM structure. This is the structure used to pass name/value element information across the AMI. Use it as follows:

```
AMELEM MyElement = {AMELEM_DEFAULT};
```

You are recommended to initialize all AMELEM structures in this way so that the *structId* and *version* fields have valid values. If the values passed for these fields are not valid, AMI will reject the structure.

It should be noted that some of the fields in this structure are string pointers that, in the default case, are set to NULL. If you wish to use these fields you must allocate the correct amount of storage prior to setting the pointer.

Next step

Now go to one of the following to continue building a C application:

- “C applications on AIX” on page 25
- “C applications on Solaris” on page 26
- “C applications on Windows” on page 27

C applications on AIX

This section explains what you have to do to prepare and run your C programs on the AIX operating system. See “Language compilers” on page 260 for compilers supported by the AMI.

Preparing C programs on AIX

The following is not prescriptive as there are many ways to set up environments to build executables. Use it as a guideline, but follow your local procedures.

To compile an AMI program in a single step using the **xlc** command you need to specify a number of options:

- Where the AMI include files are.

  This can be done using the `-I` flag. In the case of AIX, they are usually located at `/usr/mqm/amt/inc`.
- Where the AMI library is.

  This can be done using the `-L` flag. In the case of AIX, it is usually located at `/usr/mqm/lib`.
- Link with the AMI library.

  This is done with the `-l` flag, more specifically `-lamt`.

For example, compiling the C program `mine.c` into an executable called `mine`:

```
xlc -I/usr/mqm/amt/inc -L/usr/mqm/lib -lamt mine.c -o mine
```

If, however, you are building a threaded program, you must use the correct compiler and the threaded ibrary, `libamt_r.a`. For example:

```
xlc_r -I/usr/mqm/amt/inc -L/usr/mqm/lib -lamt_r mine.c -o mine
```

Running C programs on AIX

When running a C executable you must have access to the C libraries `libamt.a`, `libamtXML.a`, and `libamtICUUC.a` in your runtime environment. If the **amtInstall** utility has been run, this environment will be set up for you (see “Installation on AIX” on page 261).

If you have not run the utility, the easiest way of achieving this is to construct a link from the AIX default library location to the actual location of the C libraries. To do this:

```
  ln -s /usr/mqm/lib/libamt.a /usr/lib/libamt.a
  ln -s /usr/mqm/lib/libamtXML.a /usr/lib/libamtXML.a
  ln -s /usr/mqm/lib/libamtICUUC.a /usr/lib/libamtICUUC.a
```

You must have sufficient access to perform this operation.

If you are using the threaded libraries, you can perform a similar operation:

```
  ln -s /usr/mqm/lib/libamt_r.a /usr/lib/libamt_r.a
  ln -s /usr/mqm/lib/libamtXML_r.a /usr/lib/libamtXML_r.a
  ln -s /usr/mqm/lib/libamtICUUC_r.a /usr/lib/libamtICUUC_r.a
```

You must also make the AMI MQSeries runtime binding stubs available in your runtime environment. These stubs allow AMI to load MQSeries libraries dynamically.

For the non-threaded MQSeries Server library, perform:

```
ln -s /usr/mqm/lib/amtcmqm /usr/lib/amtcmqm
```

For the non-threaded MQSeries Client library, perform:

```
ln -s /usr/mqm/lib/amtcmqic /usr/lib/amtcmqic
```

For the threaded MQSeries Server library, perform:

```
ln -s /usr/mqm/lib/amtcmqm_r /usr/lib/amtcmqm_r
```

For the threaded MQSeries Client library, perform:

```
ln -s /usr/mqm/lib/amtcmqic_r /usr/lib/amtcmqic_r
```

C applications on Solaris

This section explains what you have to do to prepare and run your C programs in the Sun Solaris operating environment. See "Language compilers" on page 260 for compilers supported by the AMI.

Preparing C programs on Solaris

The following is not prescriptive as there are many ways to set up environments to build executables. Use it as a guideline, but follow your local procedures.

To compile an AMI program in a single step using the **CC** command you need to specify a number of options:

- Where the AMI include files are.

  This can be done using the `-I` flag. In the case of Solaris, they are usually located at `/opt/mqm/amt/inc`.
- Where the AMI library is.

  This can be done using the `-L` flag. In the case of Solaris, it is usually located at `/opt/mqm/lib`.
- Link with the AMI library.

  This is done with the `-l` flag, more specifically `-lamt`.

For example, compiling the C program `mine.c` into an executable called `mine`:

```
CC -mt -I/opt/mqm/amt/inc -L/opt/mqm/lib -lamt mine.c -o mine
```

Running C programs on Solaris

When running a C executable you must have access to the C libraries `libamt.so`, `libamtXML.so`, and `libamtICUUC.so` in your runtime environment. If the **amtInstall** utility has been run, this environment will be set up for you (see "Installation on Sun Solaris" on page 265).

If you have not run the utility, the easiest way of achieving this is to construct a link from the Solaris default library location to the actual location of the C libraries. To do this:

```
ln -s /opt/mqm/lib/libamt.so /usr/lib/libamt.so
ln -s /opt/mqm/lib/libamtXML.so /usr/lib/libamtXML.so
ln -s /opt/mqm/lib/libamtICUUC.so /usr/lib/libamtICUUC.so
```

You must have sufficient access to perform this operation.

You must also make the AMI MQSeries runtime binding stubs available in your runtime environment. These stubs allow AMI to load MQSeries libraries dynamically. For the non-threaded MQSeries Server library, perform:

```
ln -s /opt/mqm/lib/amtcmqm /usr/lib/amtcmqm
```

For the MQSeries Client library, perform:

```
ln -s /opt/mqm/lib/amtcmqic /usr/lib/amtcmqic
```

C applications on Windows

This section explains what you have to do to prepare and run your C programs on the Windows 98 and Windows NT operating systems. See "Language compilers" on page 260 for compilers supported by the AMI.

Preparing C programs on Windows

The following is not prescriptive as there are many ways to set up environments to build executables. Use it as a guideline, but follow your local procedures.

To compile an AMI program in a single step using the **cl** command you need to specify a number of options:

- Where the AMI include files are.

  This can be done using the `-I` flag. In the case of Windows, they are usually located at `\amt\include` relative to where you installed MQSeries. Alternatively, the include files could exist in one of the directories pointed to by the INCLUDE environment variable.

- Where the AMI library is.

  This can be done by including the library file `amt.LIB` as a command line argument. The `amt.LIB` file should exist in one of the directories pointed to by the LIB environment variable.

For example, compiling the C program `mine.c` into an executable called `mine.exe`:

```
cl -IC:\MQSeries\amt\include /Fomine mine.c amt.LIB
```

Running C programs on Windows

When running a C executable you must have access to the C DLLs `amt.dll` and `amtXML.dll` in your runtime environment. Make sure they exist in one of the directories pointed to by the PATH environment variable. For example:

```
SET PATH=%PATH%;C:\MQSeries\bin;
```

If you already have MQSeries installed, and you have installed AMI under the MQSeries directory structure, it is likely that the PATH has already been set up for you.

You must also make sure that your AMI runtime environment can access the MQSeries runtime environment. (This will be the case if you installed MQSeries using the documented method.)

Chapter 3. The C high-level interface

The C high-level interface contains functions that cover the requirements of the majority of applications. If extra functionality is needed, C object interface functions can be used in the same application as the C high-level functions.

This chapter contains:

Overview of the C high-level interface

The high-level functions are listed below. Follow the page references to see the detailed descriptions of each function.

Initialize and terminate

Functions to create and open an AMI session, and to close and delete an AMI session.

amInitialize	page 34
amTerminate	page 44

Sending messages

Functions to send a datagram (send and forget) message, and to send request and response messages.

amSendMsg	page 40
amSendRequest	page 41
amSendResponse	page 42

Receiving messages

Functions to receive a message from **amSendMsg** or **amSendResponse**, and to receive a request message from **amSendRequest**.

amReceiveMsg	page 36
amReceiveRequest	page 39

Publish/subscribe

Functions to publish a message to a publish/subscribe broker, and to subscribe, unsubscribe, and receive publications.

amPublish	page 35
amSubscribe	page 43
amUnsubscribe	page 45
amReceivePublication	page 37

Transaction support

Functions to commit and backout a unit of work.

amCommit	page 33
amBackout	page 32

Reference information for the C high-level interface

In the following sections the high-level interface functions are listed in alphabetical order. Note that all functions return a completion code (`pCompCode`) and a reason code (`pReason`). The completion code can take one of the following values:

AMCC_OK	Function completed successfully
AMCC_WARNING	Function completed with a warning
AMCC_FAILED	An error occurred during processing

If the completion code returns warning or failed, the reason code identifies the reason for the error or warning (see Appendix).

Most functions require the session handle to be specified. If this handle is not valid, the results are unpredictable.

amBackout

Function to backout a unit of work.

```
 AMBOOL amBackout(
   AMHSES    hSession,
   AMSTR     policyName,
   PAMLONG   pCompCode,
   PAMLONG   pReason);
```

Parameters

`hSession`	The session handle returned by **amInitialize** (input).
`policyName`	The name of a policy (input). If specified as NULL, the system default policy (constant: AMSD_POL) is used.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amCommit

Function to commit a unit of work.

```
AMBOOL amCommit(
  AMHSES    hSession,
  AMSTR     policyName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

Parameters

hSession	The session handle returned by **amInitialize** (input/output).
policyName	The name of a policy (input). If specified as NULL, the system default policy (constant: AMSD_POL) is used.
pCompCode	Completion code (output).
pReason	Reason code (output).

amInitialize

Function to create and open an AMI session. It returns a session handle of type AMHSES, which is valid until the session is terminated. One **amInitialize** is allowed per thread. A session handle can be used on different threads, subject to any limitations of the underlying transport layer (MQSeries).

```
AMHSES amInitialize(
  AMSTR     name,
  AMSTR     policyName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

Parameters

`name`	An optional name that can be used to identify the application (input).
`policyName`	The name of a policy defined in the repository (input). If specified as NULL, the system default policy name (constant: AMSD_POL) is used.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amPublish

Function to publish a message to a publish/subscribe broker.

```
AMBOOL amPublish(
  AMHSES    hSession,
  AMSTR     publisherName,
  AMSTR     policyName,
  AMSTR     receiverName,
  AMLONG    topicLen,
  AMSTR     pTopic,
  AMLONG    dataLen,
  PAMBYTE   pData,
  AMSTR     pubMsgName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

Parameters

`hSession`	The session handle returned by **amInitialize** (input).
`publisherName`	The name of a publisher service (input). If specified as NULL, the system default publisher name (constant: AMSD_PUB) is used.
`policyName`	The name of a policy (input). If specified as NULL, the system default policy name (constant: AMSD_POL) is used.
`receiverName`	The name of the receiver service to which the response to this publish request should be sent (input). Specify as NULL if no response is required. This parameter is mandatory if the policy specifies implicit publisher registration (the default).
`topicLen`	The length of the topic for this publication, in bytes (input). A value of AMLEN_NULL_TERM specifies that the string is NULL terminated.
`pTopic`	The topic for this publication (input).
`dataLen`	The length of the publication data in bytes (input). A value of zero indicates that any publication data has been added to the message object (`pubMsgName`) using the object interface (see “Message interface functions” on page 72).
`pData`	The publication data, if `dataLen` is non-zero (input).
`pubMsgName`	The name of a message object that contains the header for the publication message (input). If `dataLen` is zero it also holds any publication data. If specified as NULL, the system default message name (constant: AMSD_SND_MSG) is used.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amReceiveMsg

Function to receive a message.

```
AMBOOL amReceiveMsg(
  AMHSES    hSession,
  AMSTR     receiverName,
  AMSTR     policyName,
  AMSTR     selMsgName,
  AMLONG    buffLen,
  PAMLONG   pDataLen,
  PAMBYTE   pData,
  AMSTR     rcvMsgName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

Parameters

hSession	The session handle returned by **amInitialize** (input).
receiverName	The name of a receiver service (input). If specified as NULL, the system default receiver name (constant: AMSD_RCV) is used.
policyName	The name of a policy (input). If specified as NULL, the system default policy name (constant: AMSD_POL) is used.
selMsgName	Optional selection message object used to specify information (such as a *CorrelId*) needed to select the required message (input).
buffLen	The length in bytes of a buffer into which the data is put (input). If buffLen is set to zero and pDataLen is set to NULL, the message data is returned in the message object (rcvMsgName) instead of the pData parameter. If Accept Truncated Messages is set to 'No' in the policy options (the default), and if buffLen is set to zero but pDataLen is not NULL, the data length is returned without receiving the data. This allows the required amount of memory to be allocated before issuing a second function call to return the data. If Accept Truncated Messages is set to 'Yes' in the policy options, and either buffLen is non-zero or pDataLen is non-null, the message data might be truncated. If buffLen is zero and pDataLen is non-null, the message data is discarded.
pDataLen	The length of the message data, in bytes (output). Specify as NULL if this is not required.
pData	The received message data (output).
rcvMsgName	The name of the message object for the received message (output). Properties, and message data if not returned in the pData parameter, can be extracted from the message object using the object interface (see "Message interface functions" on page 72). The message object is implicitly reset before the receive takes place.
pCompCode	Completion code (output).
pReason	Reason code (output).

amReceivePublication

Function to receive a publication from a publish/subscribe broker.

```
AMBOOL amReceivePublication(
  AMHSES    hSession,
  AMSTR     subscriberName,
  AMSTR     policyName,
  AMSTR     selMsgName,
  AMLONG    topicBuffLen,
  AMLONG    buffLen,
  PAMLONG   pTopicCount,
  PAMLONG   pTopicLen,
  AMSTR     pFirstTopic,
  PAMLONG   pDataLen,
  PAMBYTE   pData,
  AMSTR     rcvMsgName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

Parameters

`hSession`	The session handle returned by **amInitialize** (input).
`subscriberName`	The name of a subscriber service (input). If specified as NULL, the system default subscriber name (constant: AMSD_SUB) is used.
`policyName`	The name of a policy (input). If specified as NULL, the system default policy name (constant: AMSD_POL) is used.
`selMsgName`	Optional selection message object used to specify information (such as a *CorrelId*) needed to select the required message (input).
`topicBuffLen`	The length in bytes of a buffer in which the topic is returned (input).
`buffLen`	The length in bytes of a buffer in which the publication data is returned (input).
`pTopicCount`	The number of topics in the message (output). Specify as NULL if this is not required.
`pTopicLen`	The length in bytes of the first topic (output). Specify as NULL if this is not required.
`pFirstTopic`	The first topic (output). Specify as NULL if this is not required. Topics can be extracted from the message object (`rcvMsgName`) using the object interface (see “Message interface functions” on page 72).
`pDataLen`	The length in bytes of the publication data (output). Specify as NULL if this is not required.
`pData`	The publication data (output). Specify as NULL if this is not required. Data can be extracted from the message object (`rcvMsgName`) using the object interface (see “Message interface functions” on page 72).

`rcvMsgName`	The name of a message object for the received message (input). If specified as NULL, the default message name (constant: AMSD_RCV_MSG) is used. The publication message properties and data update this message object, in addition to being returned in the parameters above. The message object is implicitly reset before the receive takes place.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amReceiveRequest

Function to receive a request message.

```
AMBOOL amReceiveRequest(
  AMHSES    hSession,
  AMSTR     receiverName,
  AMSTR     policyName,
  AMLONG    buffLen,
  PAMLONG   pDataLen,
  PAMBYTE   pData,
  AMSTR     rcvMsgName,
  AMSTR     senderName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

Parameters

`hSession`	The session handle returned by **amInitialize** (input).
`receiverName`	The name of a receiver service (input). If specified as NULL, the system default receiver name (constant: AMSD_RCV) is used.
`policyName`	The name of a policy (input). If specified as NULL, the system default policy name (constant: AMSD_POL) is used.
`buffLen`	The length in bytes of a buffer in which the data is returned (input). If `buffLen` is set to zero and `pDataLen` is set to NULL, the message data is returned in the message object (`rcvMsgName`) instead of the `pData` parameter. If Accept Truncated Messages is set to ‘No’ in the policy options (the default), and if `buffLen` is set to zero but `pDataLen` is not NULL, the data length is returned without receiving the data. This allows the required amount of memory to be allocated before issuing a second function call to return the data. If Accept Truncated Messages is set to ‘Yes’ in the policy options, and either `buffLen` is non-zero or `pDataLen` is non-null, the message data might be truncated. If `buffLen` is zero and `pDataLen` is non-null, the message data is discarded.
`pDataLen`	The length of the message data, in bytes (input/output). Specify as NULL if this is not required.
`pData`	The received message data (output).
`rcvMsgName`	The name of the message object for the received message (output). Header information, and message data if not returned in the `Data` parameter, can be extracted from the message object using the object interface (see “Message interface functions” on page 72). The message object is implicitly reset before the receive takes place.
`senderName`	The name of a special type of sender service known as a *response sender*, to which the response message will be sent (output). This sender name must not be defined in the repository.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSendMsg

Function to send a datagram (send and forget) message.

```
AMBOOL amSendMsg(
  AMHSES    hSession,
  AMSTR     senderName,
  AMSTR     policyName,
  AMLONG    dataLen,
  PAMBYTE   pData,
  AMSTR     sndMsgName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

Parameters

`hSession`	The session handle returned by **amInitialize** (input).
`senderName`	The name of a sender service (input). If specified as NULL, the system default sender name (constant: AMSD_SND) is used.
`policyName`	The name of a policy (input). If specified as NULL, the system default policy name (constant: AMSD_POL) is used.
`dataLen`	The length of the message data in bytes (input). A value of zero indicates that any message data has been added to the message object (`sndMsgName`) using the object interface (see “Message interface functions” on page 72).
`pData`	The message data, if `dataLen` is non-zero (input).
`sndMsgName`	The name of a message object for the message being sent (input). If `dataLen` is zero it also holds any message data. If specified as NULL, the system default message name (constant: AMSD_SND_MSG) is used.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSendRequest

Function to send a request message.

```
AMBOOL amSendRequest(
  AMHSES    hSession,
  AMSTR     senderName,
  AMSTR     policyName,
  AMSTR     receiverName,
  AMLONG    dataLen,
  PAMBYTE   pData,
  AMSTR     sndMsgName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

Parameters

`hSession`	The session handle returned by **amInitialize** (input).
`senderName`	The name of a sender service (input). If specified as NULL, the system default sender name (constant: AMSD_SND) is used.
`policyName`	The name of a policy (input). If specified as NULL, the system default policy (constant: AMSD_POL) is used.
`receiverName`	The name of the receiver service to which the response to this send request should be sent (input). See **amReceiveRequest**. Specify as NULL if no response is required.
`dataLen`	The length of the message data in bytes (input). A value of zero indicates that any message data has been added to the message object (`sndMsgName`) using the object interface (see “Message interface functions” on page 72).
`pData`	The message data, if `dataLen` is non-zero (input).
`sndMsgName`	The name of a message object for the message being sent (input). If specified as NULL, the system default message (constant: AMSD_SND_MSG) is used.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSendResponse

Function to send a response to a request message.

```
AMBOOL amSendResponse(
  AMHSES    hSession,
  AMSTR     senderName,
  AMSTR     policyName,
  AMSTR     rcvMsgName,
  AMLONG    dataLen,
  PAMBYTE   pData,
  AMSTR     sndMsgName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

Parameters

`hSession`	The session handle returned by **amInitialize** (input).
`senderName`	The name of the sender service (input). It must be set to the `senderName` specified for the **amReceiveRequest** function.
`policyName`	The name of a policy (input). If specified as NULL, the system default policy (constant: AMSD_POL) is used.
`rcvMsgName`	The name of the received message that this message is a response to (input). It must be set to the `rcvMsgName` specified for the **amReceiveRequest** function.
`dataLen`	The length of the message data in bytes (input). A value of zero indicates that any message data has been added to the message object (`sndMsgName`) using the object interface (see “Message interface functions” on page 72).
`pData`	The message data, if `dataLen` is non-zero (input).
`sndMsgName`	The name of a message object for the message being sent (input). If specified as NULL, the system default message (constant: AMSD_SND_MSG) is used.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSubscribe

Function to register a subscription with a publish/subscribe broker.

Publications matching the subscription are sent to the receiver service associated with the subscriber. By default, this has the same name as the subscriber service, with the addition of the suffix '.RECEIVER'.

```
AMBOOL amSubscribe(
  AMHSES    hSession,
  AMSTR     subscriberName,
  AMSTR     policyName,
  AMSTR     receiverName,
  AMLONG    topicLen,
  AMSTR     pTopic,
  AMLONG    filterLen,
  AMSTR     pFilter,
  AMSTR     subMsgName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

Parameters

`hSession`	The session handle returned by **amInitialize** (input).
`subscriberName`	The name of a subscriber service (input). If specified as NULL, the system default subscriber (constant: AMSD_SUB) is used.
`policyName`	The name of a policy (input). If specified as NULL, the system default policy (constant: AMSD_POL) is used.
`receiverName`	The name of the receiver service to which the response to this subscribe request should be sent (input). Specify as NULL if no response is required. This is not the service to which publications will be sent by the broker; they are sent to the receiver service associated with the subscriber (see above).
`topicLen`	The length of the topic for this subscription, in bytes (input).
`pTopic`	The topic for this subscription (input). Publications which match this topic, including wildcards, will be sent to the subscriber. Multiple topics can be specified in the message object (`subMsgName`) using the object interface (see "Message interface functions" on page 72).
`filterLen`	Reserved. Must be specified as `0L` (input).
`pFilter`	Reserved. Must be specified as NULL (input).
`subMsgName`	The name of a message object for the subscribe message (input). If specified as NULL, the system default message (constant: AMSD_SND_MSG) is used.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amTerminate

Closes the session, closes and deletes any implicitly created objects, and deletes the session. Any outstanding units of work are committed (if the application terminates without an **amTerminate** call being issued, any outstanding units of work are backed out).

```
AMBOOL amTerminate(
  PAMHSES   phSession,
  AMSTR     policyName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

Parameters

phSession	A *pointer* to the session handle returned by **amInitialize** (input/output).
policyName	The name of a policy (input). If specified as NULL, the system default policy (constant: AMSD_POL) is used.
pCompCode	Completion code (output).
pReason	Reason code (output).

amUnsubscribe

Function to remove a subscription from a publish/subscribe broker.

```
AMBOOL amUnsubscribe(
  AMHSES    hSession,
  AMSTR     subscriberName,
  AMSTR     policyName,
  AMSTR     receiverName,
  AMLONG    topicLen,
  AMSTR     pTopic,
  AMLONG    filterLen,
  AMSTR     pFilter,
  AMSTR     unsubMsgName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

Parameters

`hSession`	The session handle returned by **amInitialize** (input).
`subscriberName`	The name of a subscriber service (input). If specified as NULL, the system default subscriber (constant: AMSD_SUB) is used.
`policyName`	The name of a policy (input). If specified as NULL, the system default policy (constant: AMSD_POL) is used.
`receiverName`	The name of the receiver service to which the response to this unsubscribe request should be sent (input). Specify as NULL if no response is required.
`topicLen`	The length of the topic, in bytes (input).
`pTopic`	The topic that identifies the subscription which is to be removed (input). Multiple topics can be specified in the message object (`unsubMsgName`) using the object interface (see “Message interface functions” on page 72). To deregister all topics, a policy providing this option must be specified (this is not the default policy). Otherwise, to remove a previous subscription the topic information specified must match that specified on the relevant **amSubscribe** request.
`filterLen`	Reserved. Must be specified as 0L (input).
`pFilter`	Reserved. Must be specified as NULL (input).
`unsubMsgName`	The name of a message object for the unsubscribe message (input). If specified as NULL, the system default message (constant: AMSD_SND_MSG) is used.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

Chapter 4. C object interface overview

This chapter contains an overview of the structure of the C object interface. Use it to find out what functions are available in this interface.

The object interface provides sets of interface functions for each of the following objects:

Session	page 48
Message	page 50
Sender	page 52
Receiver	page 53
Distribution list	page 54
Publisher	page 55
Subscriber	page 56
Policy	page 57

These interface functions are invoked as necessary by the high-level functions. They are made available to the application programmer through this object-style interface to provide additional function where needed. An application program can mix high-level functions and object-interface functions as required.

Details of the interface functions for each object are given in the following pages. Follow the page references to see the detailed descriptions of each function.

Details of the object interface functions used by each high-level function are given on page 58.

Session interface functions

The session object creates and manages all other objects, and provides the scope for a unit of work.

Session management

Functions to create, open, close, and delete a session object.

amSesCreate	page 61
amSesOpen	page 70
amSesClose	page 61
amSesDelete	page 65

Create objects

Functions to create message, sender, receiver, distribution list, publisher, subscriber, and policy objects. Handles to these objects are returned by these functions.

amSesCreateMessage	page 62
amSesCreateSender	page 64
amSesCreateReceiver	page 63
amSesCreateDistList	page 62
amSesCreatePublisher	page 63
amSesCreateSubscriber	page 64
amSesCreatePolicy	page 62

Get object handles

Functions to get the handles for a message, sender, receiver, distribution list, publisher, subscriber, and policy objects with a specified name (needed if the objects were created implicitly by the high-level interface).

amSesGetMessageHandle	page 68
amSesGetSenderHandle	page 70
amSesGetReceiverHandle	page 69
amSesGetDistListHandle	page 67
amSesGetPublisherHandle	page 69
amSesGetSubscriberHandle	page 70
amSesGetPolicyHandle	page 69

Delete objects

Functions to delete message, sender, receiver, distribution list, publisher, subscriber, and policy objects.

amSesDeleteMessage	page 65
amSesDeleteSender	page 67
amSesDeleteReceiver	page 66
amSesDeleteDistList	page 65
amSesDeletePublisher	page 66
amSesDeleteSubscriber	page 67
amSesDeletePolicy	page 66

Transactional processing

Functions to begin, commit, and rollback a unit of work.

amSesBegin	page 60
amSesCommit	page 61
amSesRollback	page 71

Error handling

Functions to clear the error codes, and return the completion and reason codes for the last error associated with the session object.

amSesClearErrorCodes	page 60
amSesGetLastError	page 68

Message interface functions

A message object encapsulates an MQSeries message descriptor (MQMD) structure. It also contains the message data if this is not passed as a separate parameter.

Get values

Functions to get the CCSID, correlation ID, encoding, format, group status, message ID, and name of the message object.

amMsgGetCCSID	page 75
amMsgGetCorrelId	page 75
amMsgGetEncoding	page 77
amMsgGetFormat	page 77
amMsgGetGroupStatus	page 78
amMsgGetMsgId	page 79
amMsgGetName	page 79

Set values

Functions to set the coded character set ID, correlation ID, encoding, format, and group status of the message object.

amMsgSetCCSID	page 83
amMsgSetCorrelId	page 83
amMsgSetEncoding	page 84
amMsgSetFormat	page 84
amMsgSetGroupStatus	page 85

Reset values

Function to reset the message object to the state it had when first created.

amMsgReset	page 82

Read and write data

Functions to get the length of the data, get and set the data offset, and read or write byte data to or from the message object at the current offset.

amMsgGetDataLength	page 75
amMsgGetDataOffset	page 76
amMsgSetDataOffset	page 83
amMsgReadBytes	page 82
amMsgWriteBytes	page 85

Publish/subscribe topics

Functions to manipulate the topics in a publish/subscribe message.

amMsgAddTopic	page 73
amMsgDeleteTopic	page 74
amMsgGetTopic	page 81
amMsgGetTopicCount	page 81

Publish/subscribe name/value elements

Functions to manipulate the name/value elements in a publish/subscribe message.

amMsgAddElement	page 72
amMsgDeleteElement	page 73
amMsgGetElement	page 76
amMsgGetElementCount	page 76
amMsgDeleteNamedElement	page 74
amMsgGetNamedElement	page 80
amMsgGetNamedElementCount	page 80

Error handling

Functions to clear the error codes, and return the completion and reason codes from the last error associated with the message.

amMsgClearErrorCodes	page 73
amMsgGetLastError	page 78

Publish/subscribe helper macros

Helper macros provided for use with the publish/subscribe stream name and publication timestamp name/value strings.

AmMsgAddStreamName	page 86
AmMsgGetPubTimestamp	page 86
AmMsgGetStreamName	page 87

Sender interface functions

A sender object encapsulates an MQSeries object descriptor (MQOD) structure for sending a message.

Open and close

Functions to open and close the sender service.

amSndOpen	page 91
amSndClose	page 88

Send

Function to send a message.

amSndSend	page 91

Get values

Functions to get the CCSID, encoding, and name of the sender service.

amSndGetCCSID	page 89
amSndGetEncoding	page 89
amSndGetName	page 90

Error handling

Functions to clear the error codes, and return the completion and reason codes from the last error associated with the sender service.

amSndClearErrorCodes	page 88
amSndGetLastError	page 90

Receiver interface functions

A receiver object encapsulates an MQSeries object descriptor (MQOD) structure for receiving a message.

Open and close

Functions to open and close the receiver service.

amRcvOpen	page 97
amRcvClose	page 95

Receive and browse

Functions to receive or browse a message.

amRcvReceive	page 98
amRcvBrowse	page 93

Get values

Functions to get the definition type, name, and queue name of the receiver service.

amRcvGetDefnType	page 95
amRcvGetName	page 96
amRcvGetQueueName	page 97

Set values

Function to set the queue name of the receiver service.

amRcvSetQueueName	page 99

Error handling

Functions to clear the error codes, and return the completion and reason codes from the last error associated with the receiver service.

amRcvClearErrorCodes	page 95
amRcvGetLastError	page 96

Distribution list interface functions

A distribution list object encapsulates a list of sender services.

Open and close

Functions to open and close the distribution list service.

amDstOpen	page 102
amDstClose	page 100

Send

Function to send a message to the distribution list.

amDstSend	page 103

Get values

Functions to get the name of the distribution list service, a count of the sender services in the list, and a sender service handle.

amDstGetName	page 101
amDstGetSenderCount	page 101
amDstGetSenderHandle	page 102

Error handling

Functions to clear the error codes, and return the completion and reason codes from the last error associated with the distribution list.

amDstClearErrorCodes	page 100
amDstGetLastError	page 100

Publisher interface functions

A publisher object encapsulates a sender service. It provides support for publishing messages to a publish/subscribe broker.

Open and close

Functions to open and close the publisher service.

amPubOpen	page 106
amPubClose	page 104

Publish

Function to publish a message.

amPubPublish	page 107

Get values

Functions to get the CCSID, encoding, and name of the publisher service.

amPubGetCCSID	page 104
amPubGetEncoding	page 105
amPubGetName	page 106

Error handling

Functions to clear the error codes, and return the completion and reason codes from the last error associated with the publisher.

amPubClearErrorCodes	page 104
amPubGetLastError	page 105

Subscriber interface functions

A subscriber object encapsulates both a sender service and a receiver service. It provides support for subscribe and unsubscribe requests to a publish/subscribe broker, and for receiving publications from the broker.

Open and close

Functions to open and close the subscriber service.

amSubOpen	page 111
amSubClose	page 108

Broker messages

Functions to subscribe to a broker, remove a subscription, and receive publications from the broker.

amSubSubscribe	page 113
amSubUnsubscribe	page 114
amSubReceive	page 112

Get values

Functions to get the CCSID, definition type, encoding, name, and queue name of the subscriber service.

amSubGetCCSID	page 108
amSubGetDefnType	page 109
amSubGetEncoding	page 109
amSubGetName	page 110
amSubGetQueueName	page 111

Set value

Function to set the queue name of the subscriber service.

amSubSetQueueName	page 112

Error handling

Functions to clear the error codes, and return the completion and reason codes from the last error associated with the receiver.

amSubClearErrorCodes	page 108
amSubGetLastError	page 110

Policy interface functions

A policy object encapsulates details of how the message is handled (such as priority, persistence, and whether it is included in a unit of work).

Get values

Functions to get the name of the policy, and the wait time set in the policy.

amPolGetName	page 116
amPolGetWaitTime	page 116

Set value

Function to set the wait time for a receive using the policy.

amPolSetWaitTime	page 116

Error handling

Functions to clear the error codes, and return the completion and reason codes from the last error associated with the policy.

amPolClearErrorCodes	page 115
amPolGetLastError	page 115

High-level functions

Each high-level function described in Chapter 3, "The C high-level interface" on page 29 calls a number of the object interface functions, as shown below.

Table 2. Object interface calls used by the high-level functions

High-level function	Equivalent object interface calls
amBackout	amSesCreatePolicy amSesRollback
amCommit	amSesCreatePolicy amSesCommit
amInitialize	amSesCreate amSesOpen
amTerminate	amSesClose amSesDelete
amSendMsg amSendRequest amSendResponse	amSesCreateSender amSesCreatePolicy amSesCreateMessage amSndSend
amReceiveMsg amReceiveRequest	amSesCreateReceiver amSesCreatePolicy amSesCreateMessage amRcvReceive
amPublish	amSesCreatePublisher amSesCreatePolicy amSesCreateMessage amPubPublish
amSubscribe	amSesCreateSubscriber amSesCreatePolicy amSesCreateMessage amSubSubscribe
amUnsubscribe	amSesCreateSubscriber amSesCreatePolicy amSesCreateMessage amSubUnsubscribe
amReceivePublication	amSesCreateSubscriber amSesCreatePolicy amSesCreateMessage amSubReceive

If an object already exists, the appropriate call to get its handle is used instead of calling the create function again. For example, if the message object exists, **amSesGetMessageHandle** is used instead of **amSesCreateMessage**.

Chapter 5. C object interface reference

In the following sections the C object interface functions are listed by the object they refer to:

Session	page 60
Message	page 72
Sender	page 88
Receiver	page 93
Distribution list	page 100
Publisher	page 104
Subscriber	page 108
Policy	page 115

Within each section the functions are listed in alphabetical order.

Note that all functions return a completion code (`pCompCode`) and a reason code (`pReason`). The completion code can take one of the following values:

AMCC_OK	Function completed successfully
AMCC_WARNING	Function completed with a warning
AMCC_FAILED	An error occurred during processing

If the completion code returns warning or failed, the reason code identifies the reason for the error or warning (see Appendix A, "Reason codes" on page 301).

You can specify the completion code and reason code as null pointers when the function is called, in which case the value is not returned.

Most functions return AMBOOL. They return a value of AMB_TRUE if the function completed successfully, otherwise AMB_FALSE. Functions that do not return AMBOOL return a handle as specified in the following sections.

Most functions require a handle to the object they reference. If this handle is not valid, the results are unpredictable.

Session interface functions

A *session* object provides the scope for a unit of work and creates and manages all other objects, including at least one connection object. Each (MQSeries) connection object encapsulates a single MQSeries queue manager connection. The session object definition specifying the required queue manager connection can be provided by a repository policy definition and the local host file, or the local host file only which by default will name a single local queue manager with no repository. The session, when deleted, is responsible for releasing memory by closing and deleting all other objects that it manages.

Note that you should not mix MQSeries MQCONN or MQDISC requests on the same thread as AMI calls, otherwise premature disconnection might occur.

amSesBegin

Begins a unit of work, allowing an AMI application to take advantage of the resource coordination provided in MQSeries Version 5. The unit of work can subsequently be committed by **amSesCommit**, or backed out by **amSesRollback**. It should be used only when MQSeries is the transaction coordinator. If an external transaction coordinator (for example, CICS or Tuxedo) is being used, the API of the external coordinator should be used instead.

```
AMBOOL amSesBegin(
  AMHSES    hSess,
  AMHPOL    hPolicy,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

hSess	The session handle returned by **amSesCreate** (input).
hPolicy	The handle of a policy (input). If specified as AMH_NULL_HANDLE, the system default policy (constant: AMSD_POL) is used.
pCompCode	Completion code (output).
pReason	Reason code (output).

amSesClearErrorCodes

Clears the error codes in the session object.

```
AMBOOL amSesClearErrorCodes(
  AMHSES    hSess,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

hSess	The session handle returned by **amSesCreate** (input).
pCompCode	Completion code (output).
pReason	Reason code (output).

amSesClose

Closes the session object and all open objects owned by the session, and disconnects from the underlying message transport (MQSeries).

```
AMBOOL amSesClose(
  AMHSES    hSess,
  AMHPOL    hPolicy,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

hSess	The session handle returned by **amSesCreate** (input).
hPolicy	The handle of a policy (input). If specified as AMH_NULL_HANDLE, the system default policy (constant: AMSD_POL) is used.
pCompCode	Completion code (output).
pReason	Reason code (output).

amSesCommit

Commits a unit of work that was started by **amSesBegin**, or by sending or receiving a message under syncpoint control as defined in the policy options for the send or receive request.

```
AMBOOL amSesCommit(
  AMHSES    hSess,
  AMHPOL    hPolicy,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

hSess	The session handle returned by **amSesCreate** (input).
hPolicy	The handle of a policy (input). If specified as AMH_NULL_HANDLE, the system default policy (constant: AMSD_POL) is used.
pCompCode	Completion code (output).
pReason	Reason code (output).

amSesCreate

Creates the session and system default objects. **amSesCreate** returns the handle of the session object (of type AMHSES). This must be specified by other session function calls.

```
AMHSES amSesCreate(
  AMSTR     name,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

name	An optional session name that can be used to identify the application from which a message is sent (input).
pCompCode	Completion code (output).
pReason	Reason code (output).

amSesCreateDistList

Creates a distribution list object. A distribution list handle (of type AMHDST) is returned.

```
AMHDST amSesCreateDistList(
  AMHSES    hSess,
  AMSTR     name,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

hSess	The session handle returned by **amSesCreate** (input).
name	The name of the distribution list (input). This must match the name of a distribution list defined in the repository.
pCompCode	Completion code (output).
pReason	Reason code (output).

amSesCreateMessage

Creates a message object. A message handle (of type AMHMSG) is returned.

```
AMHMSG amSesCreateMessage(
  AMHSES    hSess,
  AMSTR     name,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

hSess	The session handle returned by **amSesCreate** (input).
name	The name of the message (input). This can be any name that is meaningful to the application. It is specified so that this message object can be used with the high-level interface.
pCompCode	Completion code (output).
pReason	Reason code (output).

amSesCreatePolicy

Creates a policy object. A policy handle (of type AMHPOL) is returned.

```
AMHPOL amSesCreatePolicy(
  AMHSES    hSess,
  AMSTR     name,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

hSess	The session handle returned by **amSesCreate** (input).
name	The name of the policy (input). If it matches a policy defined in the repository, the policy will be created using the repository definition, otherwise it will be created with default values. If a repository is being used and the named policy is not found in the repository, a completion code of AMCC_WARNING is returned with a reason code of AMRC_POLICY_NOT_IN_REPOS.
pCompCode	Completion code (output).

`pReason`	Reason code (output).

amSesCreatePublisher

Creates a publisher object. A publisher handle (of type AMHPUB) is returned.

```
AMHPUB amSesCreatePublisher(
  AMHSES    hSess,
  AMSTR     name,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSess`	The session handle returned by **amSesCreate** (input).
`name`	The name of the publisher (input). If it matches a publisher defined in the repository, the publisher will be created using the repository definition, otherwise it will be created with default values (that is, with a sender service name that matches the publisher name). If a repository is being used and the named publisher is not found in the repository, a completion code of AMCC_WARNING is returned with a reason code of AMRC_PUBLISHER_NOT_IN_REPOS.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSesCreateReceiver

Creates a receiver service object. A receiver handle (of type AMHRCV) is returned.

```
AMHRCV amSesCreateReceiver(
  AMHSES    hSess,
  AMSTR     name,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSess`	The session handle returned by **amSesCreate** (input).
`name`	The name of the receiver service (input). If it matches a receiver defined in the repository, the receiver will be created using the repository definition, otherwise it will be created with default values (that is, with a queue name that matches the receiver name). If a repository is being used and the named receiver is not found in the repository, a completion code of AMCC_WARNING is returned with a reason code of AMRC_RECEIVER_NOT_IN_REPOS.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSesCreateSender

Creates a sender service object. A sender handle (of type AMHSND) is returned.

```
AMHSND amSesCreateSender(
  AMHSES    hSess,
  AMSTR     name,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSess`	The session handle returned by **amSesCreate** (input).
`name`	The name of the sender service (input). If it matches a sender defined in the repository, the sender will be created using the repository definition, otherwise it will be created with default values (that is, with a queue name that matches the sender name). If a repository is being used and the named sender is not found in the repository, a completion code of AMCC_WARNING is returned with a reason code of AMRC_SENDER_NOT_IN_REPOS.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSesCreateSubscriber

Creates a subscriber object. A subscriber handle (of type AMHSUB) is returned.

```
AMHSUB amSesCreateSubscriber(
  AMHSES    hSess,
  AMSTR     name,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSess`	The session handle returned by **amSesCreate** (input).
`name`	The name of the subscriber (input). If it matches a subscriber defined in the repository, the subscriber will be created using the repository definition, otherwise it will be created with default values (that is, with a sender service name that matches the subscriber name, and a receiver service name that is the same with the addition of the suffix '.RECEIVER'). If a repository is being used and the named subscriber is not found in the repository, a completion code of AMCC_WARNING is returned with a reason code of AMRC_SUBSCRIBER_NOT_IN_REPOS.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSesDelete

Deletes the session object. Performs an implicit close if the session is open. This closes and deletes the session and all objects owned by it.

```
AMBOOL amSesDelete(
  PAMHSES   phSess,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`phSess`	A *pointer* to the session handle returned by **amSesCreate** (input/output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSesDeleteDistList

Deletes a distribution list object, and performs an implicit close if the distribution list is open.

```
AMBOOL amSesDeleteDistList(
  AMHSES    hSess,
  PAMHDST   phDistList,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSess`	The session handle returned by **amSesCreate** (input).
`phDistList`	A *pointer* to the distribution list handle (input/output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSesDeleteMessage

Deletes a message object.

```
AMBOOL amSesDeleteMessage(
  AMHSES    hSess,
  PAMHMSG   phMsg,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSess`	The session handle returned by **amSesCreate** (input).
`phMsg`	A *pointer* to the message handle (input/output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSesDeletePolicy

Deletes a policy object.

```
AMBOOL amSesDeletePolicy(
  AMHSES    hSess,
  PAMHPOL   phPolicy,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSess`	The session handle returned by **amSesCreate** (input).
`phPolicy`	A *pointer* to the policy handle (input/output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSesDeletePublisher

Deletes a publisher object, and performs an implicit close if the publisher is open.

```
AMBOOL amSesDeletePublisher(
  AMHSES    hSess,
  PAMHPUB   phPub,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSess`	The session handle returned by **amSesCreate** (input).
`phPub`	A *pointer* to the publisher handle (input/output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSesDeleteReceiver

Deletes a receiver object, and performs an implicit close if the receiver is open.

```
AMBOOL amSesDeleteReceiver(
  AMHSES    hSess,
  PAMHRCV   phReceiver,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSess`	The session handle returned by **amSesCreate** (input).
`phReceiver`	A *pointer* to the receiver service handle (input/output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSesDeleteSender

Deletes a sender object, and performs an implicit close if the sender is open.

```
AMBOOL amSesDeleteSender(
  AMHSES    hSess,
  PAMHSND   phSender,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

hSess	The session handle returned by **amSesCreate** (input).
phSender	A *pointer* to the sender service handle (input/output).
pCompCode	Completion code (output).
pReason	Reason code (output).

amSesDeleteSubscriber

Deletes a subscriber object, and performs an implicit close if the subscriber is open.

```
AMBOOL amSesDeleteSubscriber(
  AMHSES    hSess,
  PAMHSUB   phSub,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

hSess	The session handle returned by **amSesCreate** (input).
phSub	A *pointer* to the subscriber handle (input/output).
pCompCode	Completion code (output).
pReason	Reason code (output).

amSesGetDistListHandle

Returns the handle of the distribution list object (of type AMHDST) with the specified name.

```
AMHDST amSesGetDistListHandle(
  AMHSES    hSess,
  AMSTR     name,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

hSess	The session handle returned by the **amSesCreate** function (input).
name	The name of the distribution list (input).
pCompCode	Completion code (output).
pReason	Reason code (output).

amSesGetLastError

Gets the information (completion and reason codes) from the last error for the session.

```
AMBOOL amSesGetLastError(
  AMHSES    hSess,
  AMLONG    buffLen,
  PAMLONG   pStringLen,
  AMSTR     pErrorText,
  PAMLONG   pReason2,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSess`	The session handle returned by **amSesCreate** (input).
`buffLen`	Reserved, must be zero (input).
`pStringLen`	Reserved, must be NULL (input).
`pErrorText`	Reserved, must be NULL (input).
`pReason2`	A secondary reason code (output). Not returned if specified as NULL. If `pReason` indicates AMRC_TRANSPORT_WARNING or AMRC_TRANSPORT_ERR, `pReason2` gives an MQSeries reason code.
`pCompCode`	Completion code (output). Not returned if specified as NULL.
`pReason`	Reason code (output). Not returned if specified as NULL. A value of AMRC_SESSION_HANDLE_ERR indicates that the **amSesGetLastError** function call has itself detected an error and failed.

amSesGetMessageHandle

Returns the handle of the message object (of type AMHMSG) with the specified name.

```
AMHMSG amSesGetMessageHandle(
  AMHSES    hSess,
  AMSTR     name,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSess`	The session handle returned by **amSesCreate** (input).
`name`	The name of the message (input).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSesGetPolicyHandle

Returns the handle of the policy object (of type AMHPOL) with the specified name.

```
AMHPOL amSesGetPolicyHandle(
  AMHSES    hSess,
  AMSTR     name,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSess`	The session handle returned by **amSesCreate** (input).
`name`	The name of the policy (input).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSesGetPublisherHandle

Returns the handle of the publisher object (of type AMHPUB) with the specified name.

```
AMHPUB amSesGetPublisherHandle(
  AMHSES    hSess,
  AMSTR     name,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSess`	The session handle returned by **amSesCreate** (input).
`name`	The name of the publisher (input).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSesGetReceiverHandle

Returns the handle of the receiver service object (of type AMHRCV) with the specified name.

```
AMHRCV amSesGetReceiverHandle(
  AMHSES    hSess,
  AMSTR     name,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSess`	The session handle returned by **amSesCreate** (input).
`name`	The name of the receiver service (input).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSesGetSenderHandle

Returns the handle of the sender service object (of type AMHSND) with the specified name.

```
AMHSND amSesGetSenderHandle(
  AMHSES    hSess,
  AMSTR     name,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

hSess	The session handle returned by **amSesCreate** (input).
name	The name of the sender service (input).
pCompCode	Completion code (output).
pReason	Reason code (output).

amSesGetSubscriberHandle

Returns the handle of the subscriber object (of type AMHSUB) with the specified name.

```
AMHSUB amSesGetSubscriberHandle(
  AMHSES    hSess,
  AMSTR     name,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

hSess	The session handle returned by **amSesCreate** (input).
name	The name of the subscriber (input).
pCompCode	Completion code (output).
pReason	Reason code (output).

amSesOpen

Opens the session object using the specified policy options. The policy, together with the local host file, provides the connection definition that enables the connection object to be created. The specified library is loaded and initialized. If the policy connection type is specified as AUTO and the MQSeries local queue manager library cannot be loaded, the MQSeries client library is loaded. The connection to the underlying message transport (MQSeries) is then opened.

```
AMBOOL amSesOpen(
  AMHSES    hSess,
  AMHPOL    hPolicy,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

hSess	The session handle returned by **amSesCreate** (input).
hPolicy	The handle of a policy (input). If specified as AMH_NULL_HANDLE, system default policy (constant: AMSD_POL) is used.
pCompCode	Completion code (output).
pReason	Reason code (output).

amSesRollback

Rolls back a unit of work.

```
 AMBOOL amSesRollback(
   AMHSES    hSess,
   AMHPOL    hPolicy,
   PAMLONG   pCompCode,
   PAMLONG   pReason);
```

`hSess`	The session handle returned by **amSesCreate** (input).
`hPolicy`	The handle of a policy (input). If specified as AMH_NULL_HANDLE, the system default policy (constant: AMSD_POL) is used.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

Message interface functions

A *message* object encapsulates an MQSeries message descriptor (MQMD), and name/value elements such as the topic data for publish/subscribe messages. It can also contain the message data, or this can be passed as a separate parameter.

A name/value element in a message object is held in an AMELEM structure. See "Using name/value elements" on page 20 for details.

The initial state of the message object is:

`CCSID`	default queue manager CCSID
`correlationId`	all zeroes
`dataLength`	zero
`dataOffset`	zero
`elementCount`	zero
`encoding`	AMENC_NATIVE
`format`	AMFMT_STRING
`groupStatus`	AMGRP_MSG_NOT_IN_GROUP
`topicCount`	zero

When a message object is used to send a message, it will not normally be left in the same state as it was prior to the send. Therefore, if you use the message object for repeated send operations, it is advisable to reset it to its initial state (see **amMsgReset** on page 82) and rebuild it each time.

amMsgAddElement

Adds a name/value element to a message.

```
AMBOOL amMsgAddElement(
  AMHMSG    hMsg,
  PAMELEM   pElem,
  AMLONG    options,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).
`pElem`	A pointer to an AMELEM element structure, which specifies the element to be added (input). It will not replace an existing element with the same name.
`options`	A reserved field, which must be set to zero (input).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amMsgAddTopic

Adds a topic to a publish/subscribe message.

```
AMBOOL amMsgAddTopic(
  AMHMSG    hMsg,
  AMLONG    topicLen,
  AMSTR     pTopic,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).
`topicLen`	The length in bytes of the topic (input). A value of AMLEN_NULL_TERM specifies that the string is NULL terminated.
`pTopic`	The topic to be added (input).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amMsgClearErrorCodes

Clears the error codes in the message object.

```
AMBOOL amMsgClearErrorCodes(
  AMHMSG    hMsg,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amMsgDeleteElement

Deletes an element with the specified index from a message. Indexing is within all elements of the message, and might include topics (which are specialized elements).

```
AMBOOL amMsgDeleteElement(
  AMHMSG    hMsg,
  AMLONG    elemIndex,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).
`elemIndex`	The index of the required element in the message, starting from zero (input). On completion, elements with higher `elemIndex` values than that specified will have their index value reduced by one. **amMsgGetElementCount** gets the number of elements in the message.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amMsgDeleteNamedElement

Deletes a named element from a message, at the specified index. Indexing is within all elements that share the same name.

```
AMBOOL amMsgDeleteNamedElement(
  AMHMSG    hMsg,
  AMLONG    nameIndex,
  AMLONG    nameLen,
  AMSTR     pName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).
`nameIndex`	The index of the required named element in the message (input). Specifying an index of zero deletes the *first* element with the specified name. On completion, elements with higher `nameIndex` values than that specified will have their index value reduced by one. **amMsgGetNamedElementCount** gets the number of elements in the message with the specified name.
`nameLen`	The length of the element name, in bytes (input). A value of AMLEN_NULL_TERM specifies that the string is NULL terminated.
`pName`	The name of the element to be deleted (input).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amMsgDeleteTopic

Deletes a topic from a publish/subscribe message, at the specified index. Indexing is within all topics in the message.

```
AMBOOL amMsgDeleteTopic(
  AMHMSG    hMsg,
  AMLONG    topicIndex,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).
`topicIndex`	The index of the required topic in the message, starting from zero (input). **amMsgGetTopicCount** gets the number of topics in the message.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amMsgGetCCSID

Gets the coded character set identifier of the message.

```
AMBOOL amMsgGetCCSID(
  AMHMSG    hMsg,
  PAMLONG   pCCSID,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).
`pCCSID`	The coded character set identifier (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amMsgGetCorrelId

Gets the correlation identifier of the message.

```
AMBOOL amMsgGetCorrelId(
  AMHMSG    hMsg,
  AMLONG    buffLen,
  PAMLONG   pCorrelIdLen,
  PAMBYTE   pCorrelId,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).
`buffLen`	The length in bytes of a buffer in which the correlation identifier is returned (input).
`pCorrelIdLen`	The length of the correlation identifier, in bytes (output). If specified as NULL, the length is not returned.
`pCorrelId`	The correlation identifier (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amMsgGetDataLength

Gets the length of the message data in the message object.

```
AMBOOL amMsgGetDataLength(
  AMHMSG    hMsg,
  PAMLONG   pLength,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).
`pLength`	The length of the message data, in bytes (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amMsgGetDataOffset

Gets the current offset in the message data for reading or writing data bytes.

```
AMBOOL amMsgGetDataOffset(
  AMHMSG    hMsg,
  PAMLONG   pOffset,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

hMsg	The message handle returned by **amSesCreateMessage** (input).
pOffset	The byte offset in the message data (output).
pCompCode	Completion code (output).
pReason	Reason code (output).

amMsgGetElement

Gets an element from a message.

```
AMBOOL amMsgGetElement(
  AMHMSG    hMsg,
  AMLONG    elemIndex,
  PAMELEM   pElem,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

hMsg	The message handle returned by **amSesCreateMessage** (input).
elemIndex	The index of the required element in the message, starting from zero (input). **amMsgGetElementCount** gets the number of elements in the message.
pElem	The selected element in the message (output).
pCompCode	Completion code (output).
pReason	Reason code (output).

amMsgGetElementCount

Gets the total number of elements in a message.

```
AMBOOL amMsgGetElementCount(
  AMHMSG    hMsg,
  PAMLONG   pCount,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

hMsg	The message handle returned by **amSesCreateMessage** (input).
pCount	The number of elements in the message (output).
pCompCode	Completion code (output).
pReason	Reason code (output).

amMsgGetEncoding

Gets the value used to encode numeric data types for the message.

```
AMBOOL amMsgGetEncoding(
  AMHMSG    hMsg,
  PAMLONG   pEncoding,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

hMsg	The message handle returned by **amSesCreateMessage** (input).
pEncoding	The encoding of the message (output). The following values can be returned: AMENC_NATIVE AMENC_NORMAL AMENC_NORMAL_FLOAT_390 AMENC_REVERSED AMENC_REVERSED_FLOAT_390 AMENC_UNDEFINED
pCompCode	Completion code (output).
pReason	Reason code (output).

amMsgGetFormat

Gets the format of the message.

```
AMBOOL amMsgGetFormat(
  AMHMSG    hMsg,
  AMLONG    buffLen,
  PAMLONG   pFormatLen,
  AMSTR     pFormat,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

hMsg	The message handle returned by **amSesCreateMessage** (input).
buffLen	The length in bytes of a buffer in which the format is returned (input).
pFormatLen	The length of the format, in bytes (output). If specified as NULL, the length is not returned.
pFormat	The format of the message (output). The values that can be returned include the following: AMFMT_NONE AMFMT_STRING AMFMT_RF_HEADER
pCompCode	Completion code (output).
pReason	Reason code (output).

amMsgGetGroupStatus

Gets the group status of the message. This indicates whether the message is in a group, and if it is the first, middle, last or only one in the group.

```
AMBOOL amMsgGetGroupStatus(
  AMHMSG    hMsg,
  PAMLONG   pStatus,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).

`pStatus`	The group status (output). It can take one of the following values:
	`AMGRP_MSG_NOT_IN_GROUP`
	`AMGRP_FIRST_MSG_IN_GROUP`
	`AMGRP_MIDDLE_MSG_IN_GROUP`
	`AMGRP_LAST_MSG_IN_GROUP`
	`AMGRP_ONLY_MSG_IN_GROUP`
	Alternatively, bitwise tests can be performed using the constants:
	`AMGF_IN_GROUP`
	`AMGF_FIRST`
	`AMGF_LAST`

`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amMsgGetLastError

Gets the information (completion and reason codes) from the last error for the message object.

```
AMBOOL amMsgGetLastError(
  AMHMSG    hMsg,
  AMLONG    buffLen,
  PAMLONG   pStringLen,
  AMSTR     pErrorText,
  PAMLONG   pReason2,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).
`buffLen`	Reserved, must be zero (input).
`pStringLen`	Reserved, must be NULL (input).
`pErrorText`	Reserved, must be NULL (input).
`pReason2`	A secondary reason code (output). Not returned if specified as NULL. If `pReason` indicates AMRC_TRANSPORT_WARNING or AMRC_TRANSPORT_ERR, `pReason2` gives an MQSeries reason code.
`pCompCode`	Completion code (output). Not returned if specified as NULL.

pReason	Reason code (output). Not returned if specified as NULL. A value of AMRC_MSG_HANDLE_ERR indicates that the **amMsgGetLastError** function call has itself detected an error and failed.

amMsgGetMsgId

Gets the message identifier.

```
AMBOOL amMsgGetMsgId(
  AMHMSG    hMsg,
  AMLONG    buffLen,
  PAMLONG   pMsgIdLen,
  PAMBYTE   pMsgId,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

hMsg	The message handle returned by **amSesCreateMessage** (input).
buffLen	The length in bytes of a buffer in which the message identifier is returned (input).
pMsgIdLen	The length of the message identifier, in bytes (output). If specified as NULL, the length is not returned.
pMsgId	The message identifier (output).
pCompCode	Completion code (output).
pReason	Reason code (output).

amMsgGetName

Gets the name of the message object.

```
AMBOOL amMsgGetName(
  AMHMSG    hMsg,
  AMLONG    buffLen,
  PAMLONG   pNameLen,
  AMSTR     pName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

hMsg	The message handle returned by **amSesCreateMessage** (input).
buffLen	The length in bytes of a buffer into which the name is put (input). If specified as zero, only the name length is returned.
pNameLen	The length of the name, in bytes (output). If specified as NULL, only the name is returned.
pName	The message object name (output).
pCompCode	Completion code (output).
pReason	Reason code (output).

amMsgGetNamedElement

Gets a named element from a message.

```
AMBOOL amMsgGetNamedElement(
  AMHMSG    hMsg,
  AMLONG    nameIndex,
  AMLONG    nameLen,
  AMSTR     pName,
  PAMELEM   pElem,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).
`nameIndex`	The index of the required named element in the message (input). Specifying an index of zero returns the first element with the specified name. **amMsgGetNamedElementCount** gets the number of elements in the message with the specified name.
`nameLen`	The length of the element name, in bytes (input). A value of AMLEN_NULL_TERM specifies that the string is null terminated.
`pName`	The element name (input).
`pElem`	The selected named element in the message (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amMsgGetNamedElementCount

Gets the number of elements in a message with a specified name.

```
AMBOOL amMsgGetNamedElementCount(
  AMHMSG    hMsg,
  AMLONG    nameLen,
  AMSTR     pName,
  PAMLONG   pCount,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).
`nameLen`	The length of the element name, in bytes (input). A value of AMLEN_NULL_TERM specifies that the string is null terminated.
`pName`	The specified element name (input).
`pCount`	The number of elements in the message with the specified name (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amMsgGetTopic

Gets a topic from a publish/subscribe message, at the specified index. Indexing is within all topics.

```
AMBOOL amMsgGetTopic(
  AMHMSG    hMsg,
  AMLONG    topicIndex,
  AMLONG    buffLen,
  PAMLONG   pTopicLen,
  AMSTR     pTopic,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).
`topicIndex`	The index of the required topic in the message (input). Specifying an index of zero returns the first topic. **amMsgGetTopicCount** gets the number of topics in the message.
`buffLen`	The length in bytes of a buffer in which the topic is returned (input).
`pTopicLen`	The length of the topic, in bytes (output).
`pTopic`	The topic (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amMsgGetTopicCount

Gets the total number of topics in a publish/subscribe message.

```
AMBOOL amMsgGetTopicCount(
  AMHMSG    hMsg,
  PAMLONG   pCount,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).
`pCount`	The number of topics (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amMsgReadBytes

Reads up to the specified number of data bytes from the message object, starting at the current data offset (which must be positioned before the end of the data for the read to be successful). Use **amMsgSetDataOffset** to set the data offset. **amMsgReadBytes** will advance the data offset by the number of bytes read, leaving the offset immediately after the last byte read.

```
AMBOOL amMsgReadBytes(
  AMHMSG    hMsg,
  AMLONG    readLen,
  PAMLONG   pBytesRead,
  PAMBYTE   pData,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).
`readLen`	The maximum number of bytes to be read (input). The data buffer specified by `pData` must be at least this size. The number of bytes returned is the minimum of `readLen` and the number of bytes between the data offset and the end of the data.
`pBytesRead`	The number of bytes read (output). If specified as NULL, the number is not returned.
`pData`	The read data (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amMsgReset

Resets the message object its initial state (see page 72).

```
AMBOOL amMsgReset(
  AMHMSG    hMsg,
  AMLONG    options,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).
`options`	A reserved field that must be specified as zero (input).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amMsgSetCCSID

Sets the coded character set identifier of the message.

```
AMBOOL amMsgSetCCSID(
  AMHMSG    hMsg,
  AMLONG    CCSID,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).
`CCSID`	The coded character set identifier (input).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amMsgSetCorrelId

Sets the correlation identifier of the message.

```
AMBOOL amMsgSetCorrelId(
  AMHMSG    hMsg,
  AMLONG    correlIdLen,
  PAMBYTE   pCorrelId,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).
`correlIdLen`	The length of the correlation identifier, in bytes (input).
`pCorrelId`	The correlation identifier (input). Specify as NULL (with a `correlIdLen` of 0L) to set the correlation identifier to NULL.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amMsgSetDataOffset

Sets the data offset for reading or writing byte data.

```
AMBOOL amMsgSetDataOffset(
  AMHMSG    hMsg,
  AMLONG    offset,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).
`offset`	The offset in bytes (input). Set an offset of zero to read or write from the start of the data.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amMsgSetEncoding

Sets the encoding of the data in the message.

```
AMBOOL amMsgSetEncoding(
  AMHMSG    hMsg,
  AMLONG    encoding,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).

`encoding`	The encoding of the message (input). It can take one of the following values:
	`AMENC_NATIVE`
	`AMENC_NORMAL`
	`AMENC_NORMAL_FLOAT_390`
	`AMENC_REVERSED`
	`AMENC_REVERSED_FLOAT_390`
	`AMENC_UNDEFINED`

`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amMsgSetFormat

Sets the format of the message.

```
AMBOOL amMsgSetFormat(
  AMHMSG    hMsg,
  AMLONG    formatLen,
  AMSTR     pFormat,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).
`formatLen`	The length of the format, in bytes (input). A value of AMLEN_NULL_TERM specifies that the string is NULL terminated.

`pFormat`	The format of the message (input). It can take one of the following values, or an application defined string:
	`AMFMT_NONE`
	`AMFMT_STRING`
	`AMFMT_RF_HEADER`
	If set to AMFMT_NONE, the default format for the sender will be used (if available).

`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amMsgSetGroupStatus

Sets the group status of the message. This indicates whether the message is in a group, and if it is the first, middle, last or only one in the group. Once you start sending messages in a group, you must complete the group before sending any messages that are not in the group.

If you specify AMGRP_MIDDLE_MSG_IN_GROUP or AMGRP_LAST_MSG_IN_GROUP without specifying AMGRP_FIRST_MSG_IN_GROUP, the behaviour is the same as for AMGRP_FIRST_MSG_IN_GROUP and AMGRP_ONLY_MSG_IN_GROUP respectively.

If you specify AMGRP_FIRST_MSG_IN_GROUP out of sequence, then the behavior is the same as for AMGRP_MIDDLE_MSG_IN_GROUP.

```
AMBOOL amMsgSetGroupStatus(
  AMHMSG    hMsg,
  AMLONG    status,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).

`status`	The group status (input). It can take one of the following values:
	`AMGRP_MSG_NOT_IN_GROUP`
	`AMGRP_FIRST_MSG_IN_GROUP`
	`AMGRP_MIDDLE_MSG_IN_GROUP`
	`AMGRP_LAST_MSG_IN_GROUP`
	`AMGRP_ONLY_MSG_IN_GROUP`

`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amMsgWriteBytes

Writes the specified number of data bytes into the message object, starting at the current data offset. If the data offset is not at the end of the data, existing data is overwritten. Use **amMsgSetDataOffset** to set the data offset. **amMsgWriteBytes** will advance the data offset by the number of bytes written, leaving it immediately after the last byte written.

```
AMBOOL amMsgWriteBytes(
  AMHMSG    hMsg,
  AMLONG    writeLen,
  PAMBYTE   pByteData,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).
`writeLen`	The number of bytes to be written (input).
`pByteData`	The data bytes (input).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

Message interface helper macros

The following helper macros are provided for manipulation of the name/value elements in a message object. Additional helper macros can be written as required.

AmMsgAddStreamName

Adds a name/value element for the publish/subscribe stream name.

```
AmMsgAddStreamName(
  AMHMSG    hMsg,
  AMLONG    streamNameLen,
  AMSTR     pStreamName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).
`streamNameLen`	The length of the stream name, in bytes (input).
`pStreamName`	The stream name (input).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

AmMsgGetPubTimeStamp

Gets the publication time stamp name/value element.

```
AmMsgGetPubTimeStamp(
  AMHMSG    hMsg,
  AMLONG    buffLen,
  PAMLONG   pTimestampLen,
  AMSTR     pTimestamp,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).
`buffLen`	The length in bytes of a buffer in which the publication time stamp is returned (input). Specify as zero to return only the length.
`pTimestampLen`	The length of the publication time stamp, in bytes (output). If specified as NULL, the length is not returned.
`pTimestamp`	The publication time stamp (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

AmMsgGetStreamName

Gets the name/value element for the publish/subscribe stream name.

```
 AmMsgGetStreamName(
   AMHMSG    hMsg,
   AMLONG    buffLen,
   PAMLONG   pStreamNameLen,
   AMSTR     pStreamName,
   PAMLONG   pCompCode,
   PAMLONG   pReason);
```

`hMsg`	The message handle returned by **amSesCreateMessage** (input).
`buffLen`	The length in bytes of a buffer in which the stream name is returned (input). Specify as zero to return only the length.
`pStreamNameLen`	The length of the stream name, in bytes (output). If specified as NULL, the length is not returned.
`pStreamName`	The stream name (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

Sender interface functions

A *sender* object encapsulates an MQSeries object descriptor (MQOD) structure. This represents an MQSeries queue on a local or remote queue manager. An open sender service is always associated with an open connection object (such as a queue manager connection). Support is also included for dynamic sender services (those that encapsulate model queues). The required sender service object definitions can be provided from a repository, or created without a repository definition by defaulting to the existing queue objects on the local queue manager.

The high-level functions **amSendMsg**, **amSendRequest** and **amSendResponse** call these interface functions as required to open the sender service and send a message. Additional calls are provided here to give the application program extra functionality.

A sender service object must be created before it can be opened. This is done implicitly using the high-level functions, or the **amSesCreateSender** session interface functions.

A *response* sender service is a special type of sender service used for sending a response to a request message. It must be created using the default definition, and not a definition stored in a repository (see “Services and policies” on page 279). Once created, it must not be opened until used in its correct context as a response sender when receiving a request message with **amRcvReceive** or **amReceiveRequest**. When opened, its queue and queue manager properties are modified to reflect the *ReplyTo* destination specified in the message being received. When first used in this context, the sender service becomes a response sender service.

amSndClearErrorCodes

Clears the error codes in the sender object.

```
AMBOOL amSndClearErrorCodes(
  AMHSND    hSender,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSender`	The sender handle returned by **amSesCreateSender** (input).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSndClose

Closes the sender service.

```
AMBOOL amSndClose(
  AMHSND    hSender,
  AMHPOL    hPolicy,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSender`	The sender handle returned by **amSesCreateSender** (input).

`hPolicy`	The handle of a policy (input). If specified as AMH_NULL_HANDLE, the system default policy (constant: AMSD_POL) is used.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSndGetCCSID

Gets the coded character set identifier of the sender service. A non-default value reflects the CCSID of a remote system unable to perform CCSID conversion of received messages. In this case the sender must perform CCSID conversion of the message before it is sent.

```
AMBOOL amSndGetCCSID(
  AMHSND    hSender,
  PAMLONG   pCCSID,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSender`	The sender handle returned by **amSesCreateSender** (input).
`pCCSID`	The coded character set identifier (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSndGetEncoding

Gets the value used to encode numeric data types for the sender service. A non-default value reflects the encoding of a remote system unable to convert the encoding of received messages. In this case the sender must convert the encoding of the message before it is sent.

```
AMBOOL amSndGetEncoding(
  AMHSND    hSender,
  PAMLONG   pEncoding,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSender`	The sender handle returned by **amSesCreateSender** (input).
`pEncoding`	The encoding (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSndGetLastError

Gets the information (completion and reason codes) from the last error for the sender object.

```
AMBOOL amSndGetLastError(
  AMHSND    hSender,
  AMLONG    buffLen,
  PAMLONG   pStringLen,
  AMSTR     pErrorText,
  PAMLONG   pReason2,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSender`	The sender handle returned by **amSesCreateSender** (input).
`buffLen`	Reserved, must be zero (input).
`pStringLen`	Reserved, must be NULL (input).
`pErrorText`	Reserved, must be NULL (input).
`pReason2`	A secondary reason code (output). Not returned if specified as NULL. If `pReason` indicates AMRC_TRANSPORT_WARNING or AMRC_TRANSPORT_ERR, `pReason2` gives an MQSeries reason code.
`pCompCode`	Completion code (output). Not returned if specified as NULL.
`pReason`	Reason code (output). Not returned if specified as NULL. A value of AMRC_SERVICE_HANDLE_ERR indicates that the **amSndGetLastError** function call has itself detected an error and failed.

amSndGetName

Gets the name of the sender service.

```
AMBOOL amSndGetName(
  AMHSND    hSender,
  AMLONG    buffLen,
  PAMLONG   pNameLen,
  AMSTR     pName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSender`	The sender handle returned by **amSesCreateSender** (input).
`buffLen`	The length in bytes of a buffer in which the name is returned (input). If specified as zero, only the name length is returned.
`pNameLen`	The length of the name, in bytes (output). If specified as NULL, only the name is returned.
`pName`	The name of the sender service (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSndOpen

Opens the sender service.

```
AMBOOL amSndOpen(
  AMHSND    hSender,
  AMHPOL    hPolicy,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSender`	The sender handle returned by **amSesCreateSender** (input).
`hPolicy`	The handle of a policy (input). If specified as AMH_NULL_HANDLE, the system default policy (constant: AMSD_POL) is used.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSndSend

Sends a message to the destination specified by the sender service. If the sender service is not open, it will be opened (if this action is specified in the policy options).

The message data can be passed in the message object, or as a separate parameter (this means that the data does not have to be copied into the message object prior to sending the message, which might improve performance especially if the message data is large).

```
AMBOOL amSndSend(
  AMHSND    hSender,
  AMHPOL    hPolicy,
  AMHRCV    hReceiver,
  AMHMSG    hRcvMsg,
  AMLONG    dataLen,
  PAMBYTE   pData,
  AMHMSG    hSndMsg,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSender`	The sender handle returned by **amSesCreateSender** (input).
`hPolicy`	The handle of a policy (input). If specified as AMH_NULL_HANDLE, the system default policy (constant: AMSD_POL) is used.
`hReceiver`	The handle of the receiver service to which the response to this message should be sent, if the message being sent is a request message (input). Specify as AMH_NULL_HANDLE if no response is required.
`hRcvMsg`	The handle of a received message that is being responded to, if this is a response message (input). Specify as AMH_NULL_HANDLE if this is not a response message.
`dataLen`	The length of the message data, in bytes (input). If specified as zero, any message data will be passed in the message object (`hSndMsg`).

`pData`	The message data, if `dataLen` is non-zero (input).
`hSndMsg`	The handle of a message object that specifies the properties of the message being sent (input). If `dataLen` is zero, it can also contain the message data. If specified as AMH_NULL_HANDLE, the default message object (constant: AMSD_SND_MSG) is used.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

Receiver interface functions

A *receiver* object encapsulates an MQSeries object descriptor (MQOD) structure. This represents a local MQSeries queue. An open receiver service is always associated with an open connection object, such as a queue manager connection. Support is also included for dynamic receiver services (that encapsulate model queues). The required receiver service object definitions can be provided from a repository or can be created automatically from the set of existing queue objects available on the local queue manager.

There is definition type associated with each receiver service:

```
AMDT_UNDEFINED
AMDT_TEMP_DYNAMIC
AMDT_DYNAMIC
AMDT_PREDEFINED
```

A receiver service created from a repository definition will be initially of type AMDT_PREDEFINED or AMDT_DYNAMIC. When opened, its definition type might change from AMDT_DYNAMIC to AMDT_TEMP_DYNAMIC according to the properties of its underlying queue object.

A receiver service created with default values (that is, without a repository definition) will have its definition type set to AMDT_UNDEFINED until it is opened. When opened, this will become AMDT_DYNAMIC, AMDT_TEMP_DYNAMIC, or AMDT_PREDEFINED, according to the properties of its underlying queue object.

amRcvBrowse

Browses a message.

```
AMBOOL amRcvBrowse(
  AMHRCV    hReceiver,
  AMHPOL    hPolicy,
  AMLONG    options,
  AMLONG    buffLen,
  PAMLONG   pDataLen,
  PAMBYTE   pData,
  AMHMSG    hRcvMsg,
  AMHSND    hSender,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hReceiver`	The receiver handle returned by **amSesCreateReceiver** (input).
`hPolicy`	The handle of a policy (input). If specified as AMH_NULL_HANDLE, the system default policy (constant: AMSD_POL) is used.

`options`	Options controlling the browse operation (input). Possible values are:

```
AMBRW_NEXT
AMBRW_FIRST
AMBRW_CURRENT
AMBRW_RECEIVE_CURRENT
AMBRW_DEFAULT          (AMBRW_NEXT)
AMBRW_LOCK_NEXT        (AMBRW_LOCK + AMBRW_NEXT)
AMBRW_LOCK_FIRST       (AMBRW_LOCK + AMBRW_FIRST)
AMBRW_LOCK_CURRENT     (AMBRW_LOCK + AMBRW_CURRENT)
AMBRW_UNLOCK
```

`AMBRW_RECEIVE_CURRENT` is equivalent to **amRcvReceive** for the message under the browse cursor.

Note that a locked message is unlocked by another browse or receive, even though it is not for the same message.

`buffLen`	The length in bytes of a buffer in which the data is returned (input). If `buffLen` is set to zero and `dataLen` is set to NULL, the message data is returned in the message object (`hRcvMsg`) instead of the `pData` parameter. If `buffLen` is set to zero but `pDataLen` is not NULL, the data length is returned without receiving the data. This allows the required amount of memory to be allocated before issuing a second function call to return the data.
`pDataLen`	The length of the message data, in bytes (output). If specified as NULL, the data length is not returned.
`pData`	The received message data (output).
`hRcvMsg`	The handle of the message object for the received message (output).
`hSender`	The handle of the response sender service that the response message must be sent to, if this is a request message (output). This sender service must be created without a repository definition, and used exclusively for sending a response. Its definition type must be AMDT_UNDEFINED (it will be set to AMDT_RESPONSE by this call).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amRcvClearErrorCodes

Clears the error codes in the receiver service object.

```
AMBOOL amRcvClearErrorCodes(
  AMHRCV    hReceiver,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hReceiver`	The receiver handle returned by **amSesCreateReceiver** (input).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amRcvClose

Closes the receiver service.

```
AMBOOL amRcvClose(
  AMHRCV    hReceiver,
  AMHPOL    hPolicy,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hReceiver`	The receiver handle returned by **amSesCreateReceiver** (input).
`hPolicy`	The handle of a policy (input). If specified as AMH_NULL_HANDLE, the system default policy (constant: AMSD_POL) is used.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amRcvGetDefnType

Gets the definition type of the receiver service.

```
AMBOOL amRcvGetDefnType(
  AMHRCV    hReceiver,
  PAMLONG   pType,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hReceiver`	The receiver handle returned by **amSesCreateReceiver** (input).

`pType`	The definition type (output). It can be one of the following:
	`AMDT_UNDEFINED`
	`AMDT_TEMP_DYNAMIC`
	`AMDT_DYNAMIC`
	`AMDT_PREDEFINED`
	Values other than AMDT_UNDEFINED reflect the properties of the underlying queue object.

`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amRcvGetLastError

Gets the information (completion and reason codes) from the last error for the receiver object.

```
AMBOOL amRcvGetLastError(
  AMHRCV    hReceiver,
  AMLONG    buffLen,
  PAMLONG   pStringLen,
  AMSTR     pErrorText,
  PAMLONG   pReason2,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hReceiver`	The receiver handle returned by **amSesCreateReceiver** (input).
`buffLen`	Reserved, must be zero (input).
`pStringLen`	Reserved, must be NULL (input).
`pErrorText`	Reserved, must be NULL (input).
`pReason2`	A secondary reason code (output). Not returned if specified as NULL. If `pReason` indicates AMRC_TRANSPORT_WARNING or AMRC_TRANSPORT_ERR, `pReason2` gives an MQSeries reason code.
`pCompCode`	Completion code (output). Not returned if specified as NULL.
`pReason`	Reason code (output). Not returned if specified as NULL. A value of AMRC_SERVICE_HANDLE_ERR indicates that the **amRcvGetLastError** function call has itself detected an error and failed.

amRcvGetName

Gets the name of the receiver service.

```
AMBOOL amRcvGetName(
  AMHRCV    hReceiver,
  AMLONG    buffLen,
  PAMLONG   pNameLen,
  AMSTR     pName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hReceiver`	The receiver handle returned by **amSesCreateReceiver** (input).
`buffLen`	The length in bytes of a buffer into which the name is put (input). Set it to zero to return only the name length.
`pNameLen`	The length of the name, in bytes (output). Set it to NULL to return only the name.
`pName`	The name of the receiver service (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amRcvGetQueueName

Gets the queue name of the receiver service. This is used to determine the queue name of a permanent dynamic receiver service, so that it can be recreated with the same queue name in order to receive messages in a subsequent session. (See also **amRcvSetQueueName**.)

```
AMBOOL amRcvGetQueueName(
  AMHRCV    hReceiver,
  AMLONG    buffLen,
  PAMLONG   pNameLen,
  AMSTR     pQueueName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hReceiver`	The receiver handle returned by **amSesCreateReceiver** (input).
`buffLen`	The length in bytes of a buffer in which the queue name is returned (input).
`pNameLen`	The length of the queue name, in bytes (output).
`pQueueName`	The queue name of the receiver service (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amRcvOpen

Opens the receiver service.

```
AMBOOL amRcvOpen(
  AMHRCV    hReceiver,
  AMHPOL    hPolicy,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hReceiver`	The receiver handle returned by **amSesCreateReceiver** (input).
	The handle of a policy (input). If specified as AMH_NULL_HANDLE, the system default policy (constant: AMSD_POL) is used.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amRcvReceive

Receives a message.

```
AMBOOL amRcvReceive(
  AMHRCV    hReceiver,
  AMHPOL    hPolicy,
  AMHMSG    hSelMsg,
  AMLONG    buffLen,
  PAMLONG   pDataLen,
  PAMBYTE   pData,
  AMHMSG    hRcvMsg,
  AMHSND    hSender,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hReceiver`	The receiver handle returned by **amSesCreateReceiver** (input).
`hPolicy`	The handle of a policy (input). If specified as AMH_NULL_HANDLE, the system default policy (constant: AMSD_POL) is used.
`hSelMsg`	The handle of a selection message object (input). This is used to identify the message to be received (for example, using the correlation ID). Specify as AMH_NULL_HANDLE to get the next available message with no selection.
`buffLen`	The length in bytes of a buffer in which the data is returned (input). If `buffLen` is set to zero and `pDataLen` is set to NULL, the message data is returned in the message object (`hRcvMsg`) instead of the `pData` parameter. If Accept Truncated Messages is set to 'No' in the policy options (the default), and if `buffLen` is set to zero but `pDataLen` is not NULL, the data length is returned without receiving the data. This allows the required amount of memory to be allocated before issuing a second function call to return the data. If Accept Truncated Messages is set to 'Yes' in the policy options, and either `buffLen` is non-zero or `pDataLen` is non-null, the message data might be truncated. If `buffLen` is zero and `pDataLen` is non-null, the message data is discarded.
`pDataLen`	The length of the message data, in bytes (output). If specified as NULL, the data length is not returned.
`pData`	The received message data (output).
`hRcvMsg`	The handle of the message object for the received message (output). If specified as AMH_NULL_HANDLE, the default message object (constant: AMSD_RCV_MSG) is used. The message object is reset implicitly before the receive takes place.
`hSender`	The handle of the response sender service that a response message must be sent to, if this is a request message (output). This sender service must be created without a repository definition, and used exclusively for sending a response. Its definition type must be AMDT_UNDEFINED (it will be set to AMDT_RESPONSE by this call).
`pCompCode`	Completion code (output).

`pReason`	Reason code (output).

amRcvSetQueueName

Sets the queue name of the receiver service, when this encapsulates a model queue. This can be used to specify the queue name of a recreated permanent dynamic receiver service, in order to receive messages in a session subsequent to the one in which it was created. (See also **amRcvGetQueueName**.)

```
AMBOOL amRcvSetQueueName(
  AMHRCV    hReceiver,
  AMLONG    nameLen,
  AMSTR     pQueueName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hReceiver`	The receiver handle returned by **amSesCreateReceiver** (input).
`nameLen`	The length of the queue name, in bytes (input). A value of AMLEN_NULL_TERM specifies that the string is NULL terminated.
`pQueueName`	The queue name of the receiver service (input).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

Distribution list interface functions

A *distribution list* object encapsulates a list of sender objects.

amDstClearErrorCodes

Clears the error codes in the distribution list object.

```
AMBOOL amDstClearErrorCodes(
  AMHDST    hDistList,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hDistList`	The distribution list handle returned by **amSesCreateDistList** (input).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amDstClose

Closes the distribution list.

```
AMBOOL amDstClose(
  AMHDST    hDistList,
  AMHPOL    hPolicy,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hDistList`	The distribution list handle returned by **amSesCreateDistList** (input).
`hPolicy`	The handle of a policy (input). If specified as AMH_NULL_HANDLE, the system default policy (constant: AMSD_POL) is used.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amDstGetLastError

Gets the information (completion and reason codes) from the last error in the distribution list object.

```
AMBOOL amDstGetLastError(
  AMHDST    hDistList,
  AMLONG    buffLen,
  PAMLONG   pStringLen,
  AMSTR     pErrorText,
  PAMLONG   pReason2,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hDistList`	The distribution list handle returned by **amSesCreateDistList** (input).
`buffLen`	Reserved, must be zero (input).

`pStringLen`	Reserved, must be NULL (input).
`pErrorText`	Reserved, must be NULL (input).
`pReason2`	A secondary reason code (output). Not returned if specified as NULL. If `pReason` indicates AMRC_TRANSPORT_WARNING or AMRC_TRANSPORT_ERR, `pReason2` gives an MQSeries reason code.
`pCompCode`	Completion code (output). Not returned if specified as NULL.
`pReason`	Reason code (output). Not returned if specified as NULL. A value of AMRC_SERVICE_HANDLE_ERR indicates that the **amDstGetLastError** function call has itself detected an error and failed.

amDstGetName

Gets the name of the distribution list object.

```
AMBOOL amDstGetName(
  AMHDST    hDistList,
  AMLONG    buffLen,
  PAMLONG   pNameLen,
  AMSTR     pName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hDistList`	The distribution list handle returned by **amSesCreateDistList** (input).
`buffLen`	The length in bytes of a buffer into which the name is put (input). Set it to zero to return only the name length.
`pNameLen`	The length of the name, in bytes (output). Set it to NULL to return only the name.
`pName`	The distribution list object name (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amDstGetSenderCount

Gets a count of the number of sender services in the distribution list.

```
AMBOOL amDstGetSenderCount(
  AMHDST    hDistList,
  PAMLONG   pCount,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hDistList`	The distribution list handle returned by **amSesCreateDistList** (input).
`pCount`	The number of sender services (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amDstGetSenderHandle

Returns the handle (type AMHSND) of a sender service in the distribution list object with the specified index.

```
AMHSND amDstGetSenderHandle(
  AMHDST    hDistList,
  AMLONG    handleIndex,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hDistList`	The distribution list handle returned by **amSesCreateDistList** (input).
`handleIndex`	The index of the required sender service in the distribution list (input). Specify an index of zero to return the first sender service in the list. **amDstGetSenderCount** gets the number of sender services in the distribution list.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amDstOpen

Opens the distribution list object for each of the destinations in the distribution list. The completion and reason codes returned by this function call indicate if the open was unsuccessful, partially successful, or completely successful.

```
AMBOOL amDstOpen(
  AMHDST    hDistList,
  AMHPOL    hPolicy,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hDistList`	The distribution list handle returned by **amSesCreateDistList** (input).
`hPolicy`	The handle of a policy (input). If specified as AMH_NULL_HANDLE, the system default policy (constant: AMSD_POL) is used.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amDstSend

Sends a message to each sender in the distribution list.

```
AMBOOL amDstSend(
  AMHDST    hDistList,
  AMHPOL    hPolicy,
  AMHRCV    hReceiver
  AMLONG    dataLen,
  PAMBYTE   pData,
  AMHMSG    hMsg,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hDistList`	The distribution list handle returned by **amSesCreateDistList** (input).
`hPolicy`	The handle of a policy (input). If specified as AMH_NULL_HANDLE, the system default policy (constant: AMSD_POL) is used.
`hReceiver`	The handle of the receiver service to which the response to this message should be sent, if the message being sent is a request message (input). Specify as AMH_NULL_HANDLE if no response is required.
`dataLen`	The length of the message data, in bytes (input). If set to zero, the data should be passed in the message object (`hMsg`).
`pData`	The message data (input).
`hMsg`	The handle of a message object that contains the header for the message being sent (input). If `dataLen` is zero, it should also contain the message data. If specified as AMH_NULL_HANDLE, the default message object (constant: AMSD_SND_MSG) is used.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

Publisher interface functions

A *publisher* object encapsulates a sender object. It provides support for publish messages to a publish/subscribe broker.

amPubClearErrorCodes

Clears the error codes in the publisher object.

```
AMBOOL amPubClearErrorCodes(
  AMHPUB    hPublisher,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hPublisher`	The publisher handle returned by **amSesCreatePublisher** (input).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amPubClose

Closes the publisher service.

```
AMBOOL amPubClose(
  AMHPUB    hPublisher,
  AMHPOL    hPolicy,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hPublisher`	The publisher handle returned by **amSesCreatePublisher** (input).
`hPolicy`	The handle of a policy (input). If specified as AMH_NULL_HANDLE, the system default policy (constant: AMSD_POL) is used.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amPubGetCCSID

Gets the coded character set identifier of the publisher service. A non-default value reflects the CCSID of a remote system unable to perform CCSID conversion of received messages. In this case the publisher must perform CCSID conversion of the message before it is sent.

```
AMBOOL amPubGetCCSID(
  AMHPUB    hPublisher,
  PAMLONG   pCCSID,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hPublisher`	The publisher handle returned by **amSesCreatePublisher** (input).
`pCCSID`	The coded character set identifier (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amPubGetEncoding

Gets the value used to encode numeric data types for the publisher service. A non-default value reflects the encoding of a remote system unable to convert the encoding of received messages. In this case the publisher must convert the encoding of the message before it is sent.

```
AMBOOL amPubGetEncoding(
  AMHPUB    hPublisher,
  PAMLONG   pEncoding,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hPublisher`	The publisher handle returned by **amSesCreatePublisher** (input).
`pEncoding`	The encoding (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amPubGetLastError

Gets the information (completion and reason codes) from the last error for the publisher object.

```
AMBOOL amPubGetLastError(
  AMHPUB    hPublisher,
  AMLONG    buffLen,
  PAMLONG   pStringLen,
  AMSTR     pErrorText,
  PAMLONG   pReason2,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hPublisher`	The publisher handle returned by **amSesCreatePublisher** (input).
`buffLen`	Reserved, must be zero (input).
`pStringLen`	Reserved, must be NULL (input).
`pErrorText`	Reserved, must be NULL (input).
`pReason2`	A secondary reason code (output). Not returned if specified as NULL. If `pReason` indicates AMRC_TRANSPORT_WARNING or AMRC_TRANSPORT_ERR, `pReason2` gives an MQSeries reason code.
`pCompCode`	Completion code (output). Not returned if specified as NULL.
`pReason`	Reason code (output). Not returned if specified as NULL. A value of AMRC_SERVICE_HANDLE_ERR indicates that the **amPubGetLastError** function call has itself detected an error and failed.

amPubGetName

Gets the name of the publisher service.

```
AMBOOL amPubGetName(
  AMHPUB    hPublisher,
  AMLONG    buffLen,
  PAMLONG   pNameLen,
  AMSTR     pName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hPublisher`	The publisher handle returned by **amSesCreatePublisher** (input).
`buffLen`	The length in bytes of a buffer into which the name is put (input). Set it to zero to return only the name length.
`pNameLen`	The length of the name, in bytes (output). Set it to NULL to return only the name.
`pName`	The publisher object name (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amPubOpen

Opens the publisher service.

```
AMBOOL amPubOpen(
  AMHPUB    hPublisher,
  AMHPOL    hPolicy,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hPublisher`	The publisher handle returned by **amSesCreatePublisher** (input).
`hPolicy`	The handle of a policy (input). If specified as AMH_NULL_HANDLE, the system default policy (constant: AMSD_POL) is used.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amPubPublish

Publishes a message using the publisher service.

The message data is passed in the message object. There is no option to pass it as a separate parameter as with **amSndSend** (this would not give any performance improvement because the MQRFH header has to be added to the message data prior to publishing it).

```
AMBOOL amPubPublish(
  AMHPUB    hPublisher,
  AMHPOL    hPolicy,
  AMHRCV    hReceiver,
  AMHMSG    hPubMsg,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

hPublisher	The publisher handle returned by **amSesCreatePublisher** (input).
hPolicy	The handle of a policy (input). If specified as AMH_NULL_HANDLE, the system default policy (constant: AMSD_POL) is used.
hReceiver	The handle of the receiver service to which the response to this publish request should be sent (input). Specify as AMH_NULL_HANDLE if no response is required. This parameter is mandatory if the policy specifies implicit registration of the publisher.
hPubMsg	The handle of a message object for the publication message (input). If specified as AMH_NULL_HANDLE, the default message object (constant: AMSD_SND_MSG) is used.
pCompCode	Completion code (output).
pReason	Reason code (output).

Subscriber interface functions

A *subscriber* object encapsulates both a sender object and a receiver object. It provides support for subscribe and unsubscribe requests to a publish/subscribe broker, and for receiving publications from the broker.

amSubClearErrorCodes

Clears the error codes in the subscriber object.

```
AMBOOL amSubClearErrorCodes(
  AMHSUB    hSubscriber,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSubscriber`	The subscriber handle returned by **amSesCreateSubscriber** (input).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSubClose

Closes the subscriber service.

```
AMBOOL amSubClose(
  AMHSUB    hSubscriber,
  AMHPOL    hPolicy,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSubscriber`	The subscriber handle returned by **amSesCreateSubscriber** (input).
`hPolicy`	The handle of a policy (input). If specified as AMH_NULL_HANDLE, the system default policy (constant: AMSD_POL) is used.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSubGetCCSID

Gets the coded character set identifier of the subscriber's sender service. A non-default value reflects the CCSID of a remote system unable to perform CCSID conversion of received messages. In this case the subscriber must perform CCSID conversion of the message before it is sent.

```
AMBOOL amSubGetCCSID(
  AMHSUB    hSubscriber,
  PAMLONG   pCCSID,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSubscriber`	The subscriber handle returned by **amSesCreateSubscriber** (input).

`pCCSID`	The coded character set identifier (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSubGetDefnType

Gets the definition type of the subscriber's receiver service.

```
 AMBOOL amSubGetDefnType(
   AMHSUB    hSubscriber,
   PAMLONG   pType,
   PAMLONG   pCompCode,
   PAMLONG   pReason);
```

`hSubscriber`	The subscriber handle returned by **amSesCreateSubscriber** (input).

`pType`	The definition type (output). It can be:
	`AMDT_UNDEFINED`
	`AMDT_TEMP_DYNAMIC`
	`AMDT_DYNAMIC`
	`AMDT_PREDEFINED`

`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSubGetEncoding

Gets the value used to encode numeric data types for the subscriber's sender service. A non-default value reflects the encoding of a remote system unable to convert the encoding of received messages. In this case the subscriber must convert the encoding of the message before it is sent.

```
 AMBOOL amSubGetEncoding(
   AMHSUB    hSubscriber,
   PAMLONG   pEncoding,
   PAMLONG   pCompCode,
   PAMLONG   pReason);
```

`hSubscriber`	The subscriber handle returned by **amSesCreateSubscriber** (input).
`pEncoding`	The encoding (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSubGetLastError

Gets the information (completion and reason codes) from the last error for the subscriber object.

```
AMBOOL amSubGetLastError(
  AMHSUB    hSubscriber,
  AMLONG    buffLen,
  PAMLONG   pStringLen,
  AMSTR     pErrorText,
  PAMLONG   pReason2,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSubscriber`	The subscriber handle returned by **amSesCreateSubscriber** (input).
`buffLen`	Reserved, must be zero (input).
`pStringLen`	Reserved, must be NULL (input).
`pErrorText`	Reserved, must be NULL (input).
`pReason2`	A secondary reason code (output). Not returned if specified as NULL. If `pReason` indicates AMRC_TRANSPORT_WARNING or AMRC_TRANSPORT_ERR, `pReason2` gives an MQSeries reason code.
`pCompCode`	Completion code (output). Not returned if specified as NULL.
`pReason`	Reason code (output). Not returned if specified as NULL. A value of AMRC_SERVICE_HANDLE_ERR indicates that the **amSubGetLastError** function call has itself detected an error and failed.

amSubGetName

Gets the name of the subscriber object.

```
AMBOOL amSubGetName(
  AMHSUB    hSubscriber,
  AMLONG    buffLen,
  PAMLONG   pNameLen,
  AMSTR     pName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSubscriber`	The subscriber handle returned by **amSesCreateSubscriber** (input).
`buffLen`	The length in bytes of a buffer into which the name is put (input). Set it to zero to return only the name length.
`pNameLen`	The length of the name, in bytes (output). Set it to NULL to return only the name.
`pName`	The subscriber object name (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSubGetQueueName

Gets the queue name of the subscriber's receiver service object. This can be used to determine the queue name of a permanent dynamic receiver service, so that it can be recreated with the same queue name in order to receive messages in a subsequent session. (See also **amSubSetQueueName**.)

```
AMBOOL amSubGetQueueName(
  AMHSUB    hSubscriber,
  AMLONG    buffLen,
  PAMLONG   pStringLen,
  AMSTR     pQueueName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSubscriber`	The subscriber handle returned by **amSesCreateSubscriber** (input).
`buffLen`	The length in bytes of a buffer in which the queue name is returned (input). Specify as zero to return only the length.
`pStringLen`	The length of the queue name, in bytes (output). If specified as NULL, the length is not returned.
`pQueueName`	The queue name (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSubOpen

Opens the subscriber service.

```
AMBOOL amSubOpen(
  AMHSUB    hSubscriber,
  AMHPOL    hPolicy,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSubscriber`	The subscriber handle returned by **amSesCreateSubscriber** (input).
`hPolicy`	The handle of a policy (input). If specified as AMH_NULL_HANDLE, the system default policy (constant: AMSD_POL) is used.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSubReceive

Receives a message, normally a publication, using the subscriber service. The message data, topic and other elements can be accessed using the message interface functions (see page 72).

The message data is passed in the message object. There is no option to pass it as a separate parameter as with **amRcvReceive** (this would not give any performance improvement because the MQRFH header has to be removed from the message data after receiving it).

```
AMBOOL amSubReceive(
  AMHSUB    hSubscriber,
  AMHPOL    hPolicy,
  AMHMSG    hSelMsg,
  AMHMSG    hRcvMsg,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSubscriber`	The subscriber handle returned by **amSesCreateSubscriber** (input).
`hPolicy`	The handle of a policy (input). If specified as AMH_NULL_HANDLE, the system default policy (constant: AMSD_POL) is used.
`hSelMsg`	The handle of a selection message object (input). This is used to identify the message to be received (for example, using the correlation ID). Specify as AMH_NULL_HANDLE to get the next available message with no selection.
`hRcvMsg`	The handle of the message object for the received message (output). If specified as AMH_NULL_HANDLE, the default message object (constant: AMSD_RCV_MSG) is used. The message object is reset implicitly before the receive takes place.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amSubSetQueueName

Sets the queue name of the subscriber's receiver object, when this encapsulates a model queue. This can be used to specify the queue name of a recreated permanent dynamic receiver service, in order to receive messages in a session subsequent to the one in which it was created. (See also **amRcvSetQueueName**.)

```
AMBOOL amSubSetQueueName(
  AMHSUB    hSubscriber,
  AMLONG    nameLen,
  AMSTR     pQueueName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSubscriber`	The subscriber handle returned by **amSesCreateSubscriber** (input).
`nameLen`	The length of the queue name, in bytes (input).

pQueueName	The queue name (input).
pCompCode	Completion code (output).
pReason	Reason code (output).

amSubSubscribe

Sends a subscribe message to a publish/subscribe broker using the subscriber service, to register a subscription. The topic and other elements can be specified using the message interface functions (see page 72) before sending the message.

Publications matching the subscription are sent to the receiver service associated with the subscriber. By default, this has the same name as the subscriber service, with the addition of the suffix '.RECEIVER'.

```
AMBOOL amSubSubscribe(
  AMHSUB    hSubscriber,
  AMHPOL    hPolicy,
  AMHRCV    hReceiver,
  AMHMSG    hSubMsg,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

hSubscriber	The subscriber handle returned by **amSesCreateSubscriber** (input).
hPolicy	The handle of a policy (input). If specified as AMH_NULL_HANDLE, the system default policy (constant: AMSD_POL) is used.
hReceiver	The handle of the receiver service to which the response to this subscribe request should be sent (input). Specify as AMH_NULL_HANDLE if no response is required. This is not the service to which publications will be sent by the broker; they are sent to the receiver service associated with the subscriber (see above).
hSubMsg	The handle of a message object for the subscribe message (input). If specified as AMH_NULL_HANDLE, the default message object (constant: AMSD_SND_MSG) is used.
pCompCode	Completion code (output).
pReason	Reason code (output).

amSubUnsubscribe

Sends an unsubscribe message to a publish/subscribe broker using the subscriber service, to deregister a subscription. The topic and other elements can be specified using the message interface functions (see page 72) before sending the message.

To deregister all topics, a policy providing this option must be specified (this is not the default policy). Otherwise, to remove a previous subscription the topic information specified must match that specified on the relevant **amSubSubscribe** request.

```
AMBOOL amSubUnsubscribe(
  AMHSUB    hSubscriber,
  AMHPOL    hPolicy,
  AMHRCV    hReceiver,
  AMHMSG    hUnsubMsg,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hSubscriber`	The subscriber handle returned by **amSesCreateSubscriber** (input).
	The handle of a policy (input). If specified as AMH_NULL_HANDLE, the system default policy (constant: AMSD_POL) is used.
`hReceiver`	The handle of the receiver service to which the response to this unsubscribe request should be sent (input). Specify as AMH_NULL_HANDLE if no response is required.
`hUnsubMsg`	The handle of a message object for the unsubscribe message (input). If specified as AMH_NULL_HANDLE, the default message object (constant: AMSD_SND_MSG) is used.
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

Policy interface functions

A *policy* object encapsulates the set of options used for each AMI request (open, close, send, receive, publish and so on). Examples are the priority and persistence of the message, and whether the message is included in a unit of work.

amPolClearErrorCodes

Clears the error codes in the policy object.

```
AMBOOL amPolClearErrorCodes(
  AMHPOL    hPolicy,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hPolicy`	The policy handle returned by **amSesCreatePolicy** (input).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amPolGetLastError

Gets the information (completion and reason codes) from the last error for the policy object.

```
AMBOOL amPolGetLastError(
  AMHPOL    hPolicy,
  AMLONG    buffLen,
  PAMLONG   pStringLen,
  AMSTR     pErrorText,
  PAMLONG   pReason2,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hPolicy`	The policy handle returned by **amSesCreatePolicy** (input).
`buffLen`	Reserved, must be zero (input).
`pStringLen`	Reserved, must be NULL (input).
`pErrorText`	Reserved, must be NULL (input).
`pReason2`	A secondary reason code (output). Not returned if specified as NULL. If `pReason` indicates AMRC_TRANSPORT_WARNING or AMRC_TRANSPORT_ERR, `pReason2` gives an MQSeries reason code.
`pCompCode`	Completion code (output). Not returned if specified as NULL.
`pReason`	Reason code (output). Not returned if specified as NULL. A value of AMRC_POLICY_HANDLE_ERR indicates that the **amPolGetLastError** function call has itself detected an error and failed.

amPolGetName

Returns the name of the policy object.

```
AMBOOL amPolGetName(
  AMHPOL    hPolicy,
  AMLONG    buffLen,
  PAMLONG   pNameLen,
  AMSTR     pName,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hPolicy`	The policy handle returned by **amSesCreatePolicy** (input).
`buffLen`	The length in bytes of a buffer into which the name is put (input). Set it to zero to return only the name length.
`pNameLen`	The length of the name, in bytes (output). Set it to NULL to return only the name.
`pName`	The policy object name (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amPolGetWaitTime

Returns the wait time (in ms) set for this policy.

```
AMBOOL amPolGetWaitTime(
  AMHPOL    hPolicy,
  PAMLONG   pWaitTime,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hPolicy`	The policy handle returned by **amSesCreatePolicy** (input).
`pWaitTime`	The wait time, in ms (output).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

amPolSetWaitTime

Sets the wait time for any receive function using this policy.

```
AMBOOL amPolSetWaitTime(
  AMHPOL    hPolicy,
  AMLONG    waitTime,
  PAMLONG   pCompCode,
  PAMLONG   pReason);
```

`hPolicy`	The policy handle returned by **amSesCreatePolicy** (input).
`waitTime`	The wait time (in ms) to be set in the policy (input).
`pCompCode`	Completion code (output).
`pReason`	Reason code (output).

Part 3. The C++ interface

This part contains:

Chapter 6. Using the Application Messaging Interface in C++

The Application Messaging Interface for C++ (amCpp) provides a C++ style of programming, while being consistent with the object-style interface of the Application Messaging Interface for C.

This chapter describes the following:

- "Structure of the AMI"
- "Writing applications in C++" on page 121
- "Building C++ applications" on page 130

Note that the term *object* is used in this book in the object-oriented programming sense, not in the sense of MQSeries 'objects' such as channels and queues.

Structure of the AMI

The following classes are provided:

Base classes

AmSessionFactory	Creates AmSession objects.
AmSession	Creates objects within the AMI session, and controls transactional support.
AmMessage	Contains the message data, message ID and correlation ID, and options that are used when sending or receiving a message (most of which come from the policy definition).
AmSender	This is a service that represents a destination (such as an MQSeries queue) to which messages are sent.
AmReceiver	This is a service that represents a source (such as an MQSeries queue) from which messages are received.
AmDistributionList	Contains a list of sender services to provide a list of destinations.
AmPublisher	Contains a sender service where the destination is a publish/subscribe broker.
AmSubscriber	Contains a sender service (to send subscribe and unsubscribe messages to a publish/subscribe broker) and a receiver service (to receive publications from the broker).
AmPolicy	Defines how the message should be handled, including items such as priority, persistence, and whether it is included in a unit of work.

Interface and helper classes

AmObject	This is an abstract class, from which the base classes listed above inherit (with the exception of AmSessionFactory).
AmElement	This encapsulates name/value pairs for use in publish/subscribe applications.
AmStatus	This encapsulates the error status of amCpp objects.
AmString	This encapsulates string data.
AmBytes	This encapsulates binary/byte data.

Exception classes

AmException	This is the base Exception class for amCpp; all other amCpp Exceptions inherit from this class.
AmErrorException	An Exception of this type is raised when an amCpp object experiences an error with a severity level of FAILED (CompletionCode = AMCC_FAILED).
AmWarningException	An Exception of this type is raised when an amCpp object experiences an error with a severity level of WARNING (CompletionCode = AMCC_WARNING), provided that warnings have been enabled using the **enableWarnings** method.

Using the repository

You can run AMI applications with or without a repository. If you don't have a repository, you can create an object by specifying its name in a method. It will be created using the appropriate system provided definition (see "System provided definitions" on page 280).

If you have a repository, and you specify the name of an object in a method that matches a name in the repository, the object will be created using the repository definition. (If no matching name is found in the repository, the system provided definition will be used.)

System default objects

The set of system default objects created in C is not accessible directly in C++, but the SYSTEM.DEFAULT.POLICY (constant: AMSD_POL) is used to provide default behavior when a policy is not specified. Objects with identical properties to the system default objects can be created for use in C++ using the built-in definitions (see "System provided definitions" on page 280).

Writing applications in C++

This section gives a number of examples showing how to access the Application Messaging Interface using C++.

Many of the method calls are overloaded and in some cases this results in default objects being used. One example of this is the AmPolicy object which can be passed on many of the methods. For example:

Method overloading

```
mySender->send(*mySendMessage, *myPolicy);

mySender->send(*mySendMessage);
```

If a policy has been created to provide specific send behavior, use the first example. However, if the default policy is acceptable, use the second example.

The defaulting of behavior using method overloading is used throughout the examples.

Creating and opening objects

Before using the AMI, you must create and open the required objects. Objects are created with names, which might correspond to named objects in the repository. In the case of the creation of a response sender (`myResponder`) in the example below, the default name for a response type object is specified, so the object is created with default responder values.

Creating AMI objects

```
mySessionFactory = new AmSessionFactory("MY.REPOSITORY.XML");
mySession = mySessionFactory->createSession("MY.SESSION");
myPolicy = mySession->createPolicy("MY.POLICY");

mySender = mySession->createSender("AMT.SENDER.QUEUE");
myReceiver = mySession->createReceiver("AMT.RECEIVER.QUEUE");
myResponder = mySession->createSender(AMDEF_RSP_SND);

mySendMessage = mySession->createMessage("MY.SEND.MESSAGE");
myReceiveMessage = mySession->createMessage("MY.RECEIVE.MESSAGE");
```

The objects are then opened. In the following examples, the session object is opened with the default policy, whereas the sender and receiver objects are opened with a specified policy (`myPolicy`).

Opening the AMI objects

```
mySession->open();
mySender->open(*myPolicy);
myReceiver->open(*myPolicy);
```

Sending messages

The examples in this section show how to send a datagram (send and forget) message. First, the message data is written to the `mySendMessage` object. Data is always sent in byte form using the AmBytes helper class.

Writing data to a message object

```
AmBytes *dataSent = new AmBytes((const char*)"message to be sent");
mySendMessage->writeBytes(*dataSent);
```

Next, the message is sent using the sender service `mySender`.

Sending a message

```
mySender->send(*mySendMessage);
```

The policy used is either the default policy for the service, if specified, or the system default policy. The message attributes are set from the policy or service, or the default for the messaging transport.

When more control is needed you can pass a policy object:

Sending a message with a specified policy

```
mySender->send(*mySendMessage, *myPolicy);
```

The policy controls the behavior of the send command. In particular, the policy specifies whether the send is part of a unit of work, the priority, persistence and expiry of the message and whether policy components should be invoked. Whether the queue should be implicitly opened and left open can also be controlled.

To send a message to a distribution list, for instance `myDistList`, use it as the sender service:

Sending a message to a distribution list

```
myDistList->send(*mySendMessage);
```

You can set an attribute such as the *`Format`* before a message is sent, to override the default in the policy or service.

Setting an attribute in a message

```
mySendMessage->setFormat("MyFormat"):
```

Similarly, after a message has been sent you can retrieve an attribute such as the *`MessageID`*. Binary data, such as *`MessageId`* can be extracted using the AmBytes helper class.

Getting an attribute from a message

```
AmBytes msgId = mySendMessage.getMessageId();
```

For details of the message attributes that you can set and get, see “AmMessage” on page 139.

When a message object is used to send a message, it might not be left in the same state as it was prior to the send. Therefore, if you use the message object for repeated send operations, it is advisable to reset it to its initial state (see “reset” on page 164) and rebuild it each time.

Sample program

For more details, refer to the `SendAndForget.cpp` sample program (see “The sample programs” on page 277).

Receiving messages

The next example shows how to receive a message from the receiver service `myReceiver`, and to read the data from the message object `myReceiveMessage`.

Receiving a message and retrieving the data

```
myReceiver->receive(*myReceiveMessage);
AmBytes data = myReceiveMessage->readBytes(
                myReceiveMessage->getDataLength());
```

The policy used will be the default for the service if defined, or the system default policy. Greater control of the behavior of the receive can be achieved by passing a policy object.

Receiving a message with a specified policy

```
myReceiver->receive(*myReceiveMessage, *myPolicy);
```

The policy can specify the wait interval, whether the call is part of a unit of work, whether the message should be code page converted, whether all the members of a group must be there before any members can be read, and how to deal with backout failures.

To receive a specific message using its correlation ID, create a selection message object and set its *CorrelId* attribute to the required value. The selection message is then passed as a parameter on the receive.

Receiving a specific message using the correlation ID

```
AmBytes * myCorrelId = new AmBytes("MYCORRELATION");
mySelectionMessage = mySession->createMessage("MY.SELECTION.MESSAGE");
mySelectionMessage->setCorrelationId(*myCorrelId);
myReceiver->receive(*myReceiveMessage, *mySelectionMessage, *myPolicy);
```

As before, the policy is optional.

You can view the attributes of the message just received, such as the *Encoding*.

Getting an attribute from the message

```
encoding = myReceiveMessage->getEncoding();
```

Sample program

For more details, refer to the `Receiver.cpp` sample program (see "The sample programs" on page 277).

Request/response messaging

In the *request/response* style of messaging, a requester (or client) application sends a request message and expects to receive a response message back. The responder (or server) application receives the request message and produces the response message (or messages) which it sends back to the requester application. The responder application uses information in the request message to know how to send the response message back to the requester.

In the following examples 'your' refers to the responding application (the server); 'my' refers to the requesting application (the client).

The requester sends a message as described in "Sending messages" on page 122, specifying the service (`myReceiver`) to which the response message should be sent.

Sending a request message

```
mySender->send(*mySendMessage, *myReceiver);
```

A policy object can also be specified if required.

The responder receives the message as described in "Receiving messages" on page 123, using its receiver service (`yourReceiver`). It also receives details of the response service (`yourResponder`) for sending the response.

Receiving the request message

```
yourReceiver->receive(*yourReceiveMessage, *yourResponder);
```

A policy object can be specified if required, as can a selection message object (see "Receiving messages" on page 123).

The responder sends its response message (`yourReplyMessage`) to the response service, specifying the received message to which this is a response.

Sending a response to the request message

```
yourResponder->send(*yourReplyMessage, *yourReceiveMessage);
```

Finally, the requester application receives the response (`myResponseMessage`), which is correlated with the original message it sent (`mySendMessage`).

Receiving the response message

```
myReceiver->receive(*myResponseMessage, *mySendMessage);
```

In a typical application the responder might be a server operating in a loop, receiving requests and replying to them. In this case, the message objects should be set to their initial state and the data cleared before servicing the next request. This is achieved as follows:

Resetting the message object

```
yourReceiveMessage->reset();
yourResponseMessage->reset();
```

Sample programs

For more details, refer to the `Client.cpp` and `Server.cpp` sample programs (see “The sample programs” on page 277).

Publish/subscribe messaging

With *publish/subscribe* messaging a *publisher* application publishes messages to *subscriber* applications using a *broker*. The message published contains application data and one or more *topic* strings that describe the data. A subscribing application subscribes to topics informing the broker which topics it is interested in. When the broker receives a message from a publisher it compares the topics in the messages to the topics in the subscription from subscribing applications. If they match, the broker forwards the message to the subscribing application.

Data on a particular topic is published as shown in the next example.

Publishing a message on a specified topic

```
AmBytes *publicationData = new AmBytes("The weather is sunny");

myPubMessage->addTopic("Weather");
myPubMessage->writeBytes(publicationData);
myPublisher->publish(*myPubMessage, *myReceiver);
```

`myReceiver` identifies a response service to which the broker will send any response messages (indicating whether the publish was successful or not). You can also specify a policy object to modify the behavior of the command.

To subscribe to a publish/subscribe broker you need to specify one or more topics.

Subscribing to a broker on specified topics

```
mySubMessage->addTopic("Weather");
mySubMessage->addTopic("Birds");
mySubscriber->subscribe(*mySubMessage, *myReceiver);
```

Broker response messages will be sent to `myReceiver`.

To remove a subscription, add the topic or topics to be deleted to the message object, and use:

Removing a subscription

```
mySubscriber->unsubscribe(*myUnsubMessage, *myReceiver);
```

To receive a publication from a broker, use:

Receiving a publication

```
mySubscriber->receive(*myReceiveMessage, *myPolicy);
publication = myReceiveMessage->readBytes(
                *myReceiveMessage->getDataLength());
```

You can then use the **getTopicCount** and **getTopic** methods to extract the topic or topics from the message object.

Sample programs

For more details, refer to the `Publisher.cpp` and `Subscriber.cpp` sample programs (see "The sample programs" on page 277).

Using AmElement objects

Publish/subscribe brokers (such as MQSeries Publish/Subscribe) respond to messages that contain name/value pairs to define the commands and options to be carried out. The Application Messaging Interface contains some methods which produce these name/value pairs directly (such as **AmSubscriber->subscribe**). For less commonly used commands, the name/value pairs can be added to a message using an AmElement object.

For example, to send a message containing a 'Request Update' command, use the following:

Using an AmElement object to construct a command message

```
AmElement *bespokeElement = new AmElement("MQPSCommand", "ReqUpdate");
mySendMessage->addElement(*bespokeElement);
```

You must then send the message, using **AmSender->send**, to the sender service specified for your publish/subscribe broker.

If you use streams with MQSeries Publish/Subscribe, you must add the appropriate name/value element explicitly to the message object.

The message element methods can, in fact, be used to add any element to a message before issuing an publish/subscribe request. Such elements (including topics, which are specialized elements) supplement or override those added implicitly by the request, as appropriate to the individual element type.

The use of name/value elements is not restricted to publish/subscribe applications. They can be used in other applications as well.

Error handling

The **getLastErrorStatus** method always reflects the last most severe error experienced by an object. It can be used to return an AmStatus object encapsulating this error state. Once the error state has been handled, **clearErrorCodes** can be called to reset this error state.

AmCpp can raise two types of Exception, one to reflect serious errors and the other to reflect warnings. By default, only AmErrorExceptions are raised. AmWarningExceptions can be enabled using the **enableWarnings** method. Since both are types of AmException, a generic catch block can be used to process all amCpp Exceptions.

Enabling AmWarningExceptions might have some unexpected side-effects, especially when an AmObject is returning data such as another AmObject. For example, if AmWarningExceptions are enabled for an AmSession object and an AmSender is created that does not exist in the repository, an AmWarningException will be raised to reflect this fact. If this happens, the AmSender object will not be created since its creation was interrupted by an Exception. However, there might be times during the life of an AmObject when processing AmWarningExceptions is useful.

For example:

```
try
{
    ...
    mySession->enableWarnings(AMB_TRUE);
    mySession->open();
    ...
}
catch (AmErrorException &errorEx)
{
    AmStatus sessionStatus = mySession->getLastErrorStatus();
    switch (sessionStatus.getReasonCode())
    {
    case AMRC_XXXX:
        ...
    case AMRC_XXXX:
        ...
    }
    mySession->clearErrorCodes();
}
catch (AmWarningException &warningEx)
{
    ...
}
```

Since most of the objects are types of AmObject, a generic error handling routine can be written. For example:

```
try
{
    ...
    mySession->open();
    ...
    mySender->send(*myMessage):
    ...
    mySender->send(*myMessage):
    ...
    mySession->commit();
}
catch(AmException &amex);
{
    AmStatus status = amex.getSource()->getLastErrorStatus();
    printf("Object in error; name = %s\n", amex.getSource()->getName());
    printf("Object in error; RC = %ld\n",  status.getReasonCode());
    ...
    amex.getSource()->clearErrorCodes();
}
```

The catch block works because all objects that throw the AmException in the try block are AmObjects, and so they all have **getName**, **getLastErrorStatus** and **clearErrorCodes** methods.

Transaction support

Messages sent and received by the AMI can, optionally, be part of a transactional unit of work. A message is included in a unit of work based on the setting of the syncpoint attribute specified in the policy used on the call. The scope of the unit of work is the session handle and only one unit of work may be active at any time.

The API calls used to control the transaction depends on the type of transaction is being used.

- MQSeries messages are the only resource

  A transaction is started by the first message sent or received under syncpoint control, as specified in the policy specified for the send or receive. Multiple messages can be included in the same unit of work. The transaction is committed or backed out using the **commit** or **rollback** method.

- Using MQSeries as an XA transaction coordinator

  The transaction must be started explicitly using the **begin** method before the first recoverable resource (such as a relational database) is changed. The transaction is committed or backed out using an **commit** or **rollback** method.

- Using an external transaction coordinator

  The transaction is controlled using the API calls of an external transaction coordinator (such as CICS, Encina or Tuxedo). The AMI calls are not used but the syncpoint attributed must still be specified in the policy used on the call.

Other considerations

Multithreading

If you are using multithreading with the AMI, a session normally remains locked for the duration of a single AMI call. If you use receive with wait, the session remains locked for the duration of the wait, which might be unlimited (that is, until the wait time is exceeded or a message arrives on the queue). If you want another thread to run while a thread is waiting for a message, it must use a separate session.

AMI handles and object references can be used on a different thread from that on which they were first created for operations that do not involve an access to the underlying (MQSeries) message transport. Functions such as initialize, terminate, open, close, send, receive, publish, subscribe, unsubscribe, and receive publication will access the underlying transport restricting these to the thread on which the session was first opened (for example, using **AmSession->open**). An attempt to issue these on a different thread will cause an error to be returned by MQSeries and a transport error (AMRC_TRANSPORT_ERR) will be reported to the application.

Using MQSeries with the AMI

You must not mix MQSeries function calls with AMI calls within the same process.

Field limits

When string and binary properties such as queue name, message format, and correlation ID are set, the maximum length values are determined by MQSeries, the underlying message transport. See the rules for naming MQSeries objects in the *MQSeries Application Programming Guide*.

Building C++ applications

AMI include files

AMI provides include files, **amtc.h** and **amtcpp.hpp**, to assist you with the writing of your applications. It is recommended that you become familiar with the contents of these files.

The include files are installed under:

```
/amt/inc          (UNIX)

\amt\include      (Windows)
```

See "Directory structure" on page 263 (AIX), 267 (Solaris), or 270 (Windows).

Your AMI C++ program must contain the statement:

```
#include <amtcpp.hpp>
```

Even though you need mention only the C++ include file, both **amtc.h** and **amtcpp.hpp** must be accessible to your program at compilation time.

Next step

Now go to one of the following to continue building a C++ application:

- "C++ applications on AIX"
- "C++ applications on Solaris" on page 132
- "C++ applications on Windows" on page 133

C++ applications on AIX

This section explains what you have to do to prepare and run your C++ programs on the AIX operating system. See "Language compilers" on page 260 for the compilers supported by the AMI.

Preparing C++ programs on AIX

The following is not prescriptive as there are many ways to set up environments to build executables. Use it as a guideline, but follow your local procedures.

To compile an AMI program in a single step using the **xlC** command you need to specify a number of options:

- Where the AMI include files are.

  This can be done using the `-I` flag. In the case of AIX, they are usually located at `/usr/mqm/amt/inc`.

- Where the AMI library is.

  This can be done using the `-L` flag. In the case of AIX, it is usually located at `/usr/mqm/lib`.

- Link with the AMI library.

  This is done with the `-l` flag, more specifically `-lamtCpp`.

For example, compiling the C++ program `mine.cpp` into an executable called `mine`:

```
xlC -I/usr/mqm/amt/inc -L/usr/mqm/lib -lamtCpp mine.cpp -o mine
```

If, however, you are building a threaded program, you must use the correct compiler and the threaded library `libamtCpp_r.a`. For example:

```
xlC_r -I/usr/mqm/amt/inc -L/usr/mqm/lib -lamtCpp_r mine.cpp -o mine
```

Running C++ programs on AIX

When running a C++ executable you must have access to the C++ library `libamtCpp.a` in your runtime environment. If the **amtInstall** utility has been run, this environment will be set up for you (see “Installation on AIX” on page 261).

If you have not run the utility, the easiest way of achieving this is to construct a link from the AIX default library location to the actual location of the C++ library. To do this:

```
ln -s /usr/mqm/lib/libamtCpp.a /usr/lib/libamtCpp.a
```

If you are using the threaded libraries, you can perform a similar operation:

```
ln -s /usr/mqm/lib/libamtCpp_r.a /usr/lib/libamtCpp_r.a
```

You also need access to the C libraries and MQSeries in your runtime environment. This is done by making the AMI MQSeries runtime binding stubs available, to allow AMI to load MQSeries libraries dynamically. For the non-threaded MQSeries Server library, perform:

```
ln -s /usr/mqm/lib/amtcmqm /usr/lib/amtcmqm
```

For the non-threaded MQSeries Client library, perform:

```
ln -s /usr/mqm/lib/amtcmqic /usr/lib/amtcmqic
```

For the threaded MQSeries Server library, perform:

```
ln -s /usr/mqm/lib/amtcmqm_r /usr/lib/amtcmqm_r
```

For the threaded MQSeries Client library, perform:

```
ln -s /usr/mqm/lib/amtcmqic_r /usr/lib/amtcmqic_r
```

C++ applications on Solaris

This section explains what you have to do to prepare and run your C++ programs in the Sun Solaris operating environment. See "Language compilers" on page 260 for the compilers supported by the AMI.

Preparing C++ programs on Solaris

The following is not prescriptive as there are many ways to set up environments to build executables. Use it as a guideline, but follow your local procedures.

To compile an AMI program in a single step using the **CC** command you need to specify a number of options:

- Where the AMI include files are.

  This can be done using the `-I` flag. In the case of Solaris, they are usually located at `/opt/mqm/amt/inc`.

- Where the AMI library is.

  This can be done using the `-L` flag. In the case of Solaris, it is usually located at `/opt/mqm/lib`.

- Link with the AMI library.

  This is done with the `-l` flag, more specifically `-lamtCpp`.

For example, compiling the C++ program `mine.cpp` into an executable called `mine`:

```
CC -mt -I/opt/mqm/amt/inc -L/opt/mqm/lib -lamtCpp mine.cpp -o mine
```

Running C++ programs on Solaris

When running a C++ executable you must have access to the C++ library `libamtCpp.so` in your runtime environment. If the **amtInstall** utility has been run, this environment will be set up for you (see "Installation on Sun Solaris" on page 265).

If you have not run the utility, the easiest way of achieving this is to construct a link from the Solaris default library location to the actual location of the C++ libraries. To do this:

```
ln -s /opt/mqm/lib/libamtCpp.so /usr/lib/libamtCpp.so
```

You also need access to the C libraries and MQSeries in your runtime environment. This is done by making the AMI MQSeries runtime binding stubs available, to allow AMI to load MQSeries libraries dynamically. For the MQSeries Server library, perform:

```
ln -s /opt/mqm/lib/amtcmqm /usr/lib/amtcmqm
```

For the MQSeries Client library, perform:

```
ln -s /opt/mqm/lib/amtcmqic /usr/lib/amtcmqic
```

C++ applications on Windows

This section explains what you have to do to prepare and run your C++ programs on the Windows 98 and Windows NT operating systems. See "Language compilers" on page 260 for the compilers supported by the AMI.

Preparing C++ programs on Windows

The following is not prescriptive as there are many ways to set up environments to build executables. Use it as a guideline, but follow your local procedures.

To compile an AMI program in a single step using the **cl** command you need to specify a number of options:

1. Where the AMI include files are.

   This can be done using the `/I` flag. In the case of Windows, they are usually located at `\amt\include` relative to where you installed MQSeries. Alternatively, the include files could exist in one of the directories pointed to by the INCLUDE environment variable.

2. Where the AMI library is.

   This can be done by including the AMT library file `amtCpp.LIB` as a command line argument. The `amtCpp.LIB` file should exist in one of the directories pointed to by the LIB environment variable.

For example, compiling the C++ program `mine.cpp` into an executable called `mine.exe`:

```
cl -IC:\MQSeries\amt\include /Fomine mine.cpp amtCpp.LIB
```

Running C++ programs on Windows

When running a C++ executable you must have access to the C++ DLL `amtCpp.dll` in your runtime environment. Make sure it exists in one of the directories pointed to by the PATH environment variable. For example:

```
SET PATH=%PATH%;C:\MQSeries\bin;
```

If you already have MQSeries installed, and you have installed AMI under the MQSeries directory structure, it is likely that the PATH has already been set up for you.

You also need access to the C libraries and MQSeries in your runtime environment. (This will be the case if you installed MQSeries using the documented method.)

Chapter 7. C++ interface overview

This chapter contains an overview of the structure of the Application Messaging Interface for C++. Use it to find out what functions are available in this interface.

The C++ interface provides sets of methods for each of the classes listed below. The methods available for each class are listed in the following pages. Follow the page references to see the reference information for each method.

AmSessionFactory

The **AmSessionFactory** class is used to create AmSession objects.

Constructor

Constructor for AmSessionFactory.

AmSessionFactory	page 152

Session factory management

Methods to return the name of an AmSessionFactory object, to get and set the names of the AMI data files (local host and repository), and to control traces.

getFactoryName	page 152
getLocalHost	page 152
getRepository	page 152
getTraceLevel	page 153
getTraceLocation	page 153
setLocalHost	page 153
setRepository	page 153
setTraceLevel	page 153
setTraceLocation	page 153

Create and delete session

Methods to create and delete an AmSession object.

createSession	page 152
deleteSession	page 152

AmSession

The **AmSession** object creates and manages all other objects, and provides scope for a unit of work.

Session management

Methods to open and close an AmSession object, to return its name, and to control traces.

open	page 158
close	page 154
getName	page 157
getTraceLevel	page 158
getTraceLocation	page 158

Create objects

Methods to create AmMessage, AmSender, AmReceiver, AmDistributionList, AmPublisher, AmSubscriber, and AmPolicy objects.

createMessage	page 155
createSender	page 156
createReceiver	page 155
createDistributionList	page 155
createPublisher	page 155
createSubscriber	page 156
createPolicy	page 155

Delete objects

Methods to delete AmMessage, AmSender, AmReceiver, AmDistributionList, AmPublisher, AmSubscriber, and AmPolicy objects.

deleteMessage	page 156
deleteSender	page 157
deleteReceiver	page 157
deleteDistributionList	page 156
deletePublisher	page 157
deleteSubscriber	page 157
deletePolicy	page 156

Transactional processing

Methods to begin, commit and rollback a unit of work.

begin	page 154
commit	page 154
rollback	page 158

Error handling

Methods to clear the error codes, enable warnings, and return the status from the last error.

clearErrorCodes	page 154
enableWarnings	page 157
getLastErrorStatus	page 157

AmMessage

An **AmMessage** object encapsulates an MQSeries message descriptor (MQMD) structure, and contains the message data.

Get values

Methods to get the CCSID, correlation ID, encoding, format, group status, message ID and name of the message object.

getCCSID	page 161
getCorrelationId	page 161
getEncoding	page 161
getFormat	page 162
getGroupStatus	page 162
getMessageId	page 162
getName	page 162

Set values

Methods to set the CCSID, correlation ID, format and group status of the message object.

setCCSID	page 164
setCorrelationId	page 164
setEncoding	page 164
setFormat	page 165
setGroupStatus	page 165

Reset values

Method to reset the message object to the state it had when first created.

reset	page 164

Read and write data

Methods to read or write byte data to or from the message object, to get and set the data offset, and to get the length of the data.

getDataLength	page 161
getDataOffset	page 161
setDataOffset	page 164
readBytes	page 163
writeBytes	page 165

Publish/subscribe topics

Methods to manipulate the topics in a publish/subscribe message.

addTopic	page 159
deleteTopic	page 160
getTopic	page 163
getTopicCount	page 163

Publish/subscribe name/value elements

Methods to manipulate the name/value elements in a publish/subscribe message.

addElement	page 159
deleteElement	page 160
getElement	page 161
getElementCount	page 161
deleteNamedElement	page 160
getNamedElement	page 163
getNamedElementCount	page 163

Error handling

Methods to clear the error codes, enable warnings, and return the status from the last error.

clearErrorCodes	page 159
enableWarnings	page 160
getLastErrorStatus	page 162

AmSender

An **AmSender** object encapsulates an MQSeries object descriptor (MQOD) structure.

Open and close

Methods to open and close the sender service.

open	page 167
close	page 166

Send

Method to send a message.

send	page 167

Get values

Methods to get the CCSID, encoding and name of the sender service.

getCCSID	page 166
getEncoding	page 167
getName	page 167

Error handling

Methods to clear the error codes, enable warnings, and return the status from the last error.

clearErrorCodes	page 166
enableWarnings	page 166
getLastErrorStatus	page 167

AmReceiver

An **AmReceiver** object encapsulates an MQSeries object descriptor (MQOD) structure.

Open and close

Methods to open and close the receiver service.

open	page 170
close	page 169

Receive and browse

Methods to receive or browse a message.

receive	page 170
browse	page 168

Get values

Methods to get the definition type, name and queue name of the receiver service.

getDefinitionType	page 169
getName	page 170
getQueueName	page 170

Set value

Method to set the queue name of the receiver service.

setQueueName	page 170

Error handling

Methods to clear the error codes, enable warnings, and return the status from the last error.

clearErrorCodes	page 169
enableWarnings	page 169
getLastErrorStatus	page 169

AmDistributionList

An **AmDistributionList** object encapsulates a list of AmSender objects.

Open and close

Methods to open and close the distribution list service.

open	page 172
close	page 171

Send

Method to send a message to the distribution list.

send	page 172

Get values

Methods to get the name of the distribution list service, a count of the AmSenders in the list, and one of the AmSenders that is contained in the list.

getName	page 171
getSenderCount	page 172
getSender	page 171

Error handling

Methods to clear the error codes, enable warnings, and return the status from the last error.

clearErrorCodes	page 171
enableWarnings	page 171
getLastErrorStatus	page 171

AmPublisher

An **AmPublisher** object encapsulates a sender service and provides support for publishing messages to a publish/subscribe broker.

Open and close

Methods to open and close the publisher service.

open	page 174
close	page 173

Publish

Method to publish a message.

publish	page 174

Get values

Methods to get the CCSID, encoding and name of the publisher service.

getCCSID	page 173
getEncoding	page 173
getName	page 174

Error handling

Methods to clear the error codes, enable warnings, and return the status from the last error.

clearErrorCodes	page 173
enableWarnings	page 173
getLastErrorStatus	page 173

AmSubscriber

An **AmSubscriber** object encapsulates both a sender service and a receiver service. It provides support for subscribe and unsubscribe requests to a publish/subscribe broker, and for receiving publications from the broker.

Open and close

Methods to open and close the subscriber service.

open	page 176
close	page 175

Broker messages

Methods to subscribe to a broker, remove a subscription, and receive a publication from the broker.

subscribe	page 178
unsubscribe	page 178
receive	page 177

Get values

Methods to get the CCSID, definition type, encoding, name and queue name of the subscriber service.

getCCSID	page 175
getDefinitionType	page 175
getEncoding	page 176
getName	page 176
getQueueName	page 176

Set value

Method to set the queue name of the subscriber service.

setQueueName	page 177

Error handling

Methods to clear the error codes, enable warnings, and return the status from the last error.

clearErrorCodes	page 175
enableWarnings	page 175
getLastErrorStatus	page 176

AmPolicy

An **AmPolicy** object encapsulates the options used during AMI operations.

Policy management

Methods to return the name of the policy, and to get and set the wait time when receiving a message.

getName	page 179
getWaitTime	page 179
setWaitTime	page 179

Error handling

Methods to clear the error codes, enable warnings, and return the status from the last error.

clearErrorCodes	page 179
enableWarnings	page 179
getLastErrorStatus	page 179

Helper classes

The classes that encapsulate name/value elements for publish/subscribe, strings, binary data and error status.

AmBytes

The AmBytes class is an encapsulation of a byte array. It allows the AMI to pass byte strings across the interface and enables manipulation of byte strings. It contains constructors, operators and a destructor, and methods to copy, compare, and pad. AmBytes also has methods to give the length of the encapsulated bytes and a method to reference the data contained within an AmBytes object.

constructors	page 180
destructor	page 181
operators	page 181
cmp	page 180
cpy	page 181
dataPtr	page 181
length	page 181
pad	page 181

AmElement

Constructor for AmElement, and methods to return the name, type, value and version of an element, to set the version, and to return an AmString representation of the element.

AmElement	page 182
getName	page 182
getValue	page 182
getVersion	page 182
setVersion	page 182
toString	page 182

AmObject

A virtual class containing methods to return the name of the object, to clear the error codes and to return the last error condition.

clearErrorCodes	page 183
getLastErrorStatus	page 183
getName	page 183

AmStatus

Constructor for AmStatus, and methods to return the completion code, reason code, secondary reason code and status text, and to return an AmString representation of the AmStatus.

AmStatus	page 184
getCompletionCode	page 184
getReasonCode	page 184
getReasonCode2	page 184
toString	page 184

AmString

The AmString class is an encapsulation of a string. It allows the AMI to pass strings across the interface and enables manipulation of strings. It contains constructors, operators, a destructor, and methods to copy, concatenate, pad, split, truncate and strip. AmString also has methods to give the length of the encapsulated string, compare AmStrings, check whether one AmString is contained within another and a method to reference the text of an AmString.

constructors	page 185
destructor	page 186
operators	page 186
cat	page 185
cmp	page 185
contains	page 185
cpy	page 185
length	page 186
pad	page 186
split	page 186
strip	page 186
text	page 186
truncate	page 186

Exception classes

Classes that encapsulate error and warning conditions. AmErrorException and AmWarningException inherit from AmException.

AmException

Methods to return the completion code and reason code from the Exception, the class name, method name and source of the Exception, and to return a string representation of the Exception.

getClassName	page 187
getCompletionCode	page 187
getMethodName	page 187
getReasonCode	page 187
getSource	page 187
toString	page 187

AmErrorException

Methods to return the completion code and reason code from the Exception, the class name, method name and source of the Exception, and to return a string representation of the Exception.

getClassName	page 188
getCompletionCode	page 188
getMethodName	page 188
getReasonCode	page 188
getSource	page 188
toString	page 188

AmWarningException

Methods to return the completion code and reason code from the Exception, the class name, method name and source of the Exception, and to return a string representation of the Exception.

getClassName	page 189
getCompletionCode	page 189
getMethodName	page 189
getReasonCode	page 189
getSource	page 189
toString	page 189

Chapter 8. C++ interface reference

In the following sections the C++ interface methods are listed by the class they refer to. Within each section the methods are listed in alphabetical order.

Base classes

Note that all of the methods in these classes can throw AmWarningException and AmErrorException (see below). However, by default, AmWarningExceptions are not raised.

AmSessionFactory	page 152
AmSession	page 154
AmMessage	page 159
AmSender	page 166
AmReceiver	page 168
AmDistributionList	page 171
AmPublisher	page 173
AmSubscriber	page 175
AmPolicy	page 179

Helper classes

AmBytes	page 180
AmElement	page 182
AmObject	page 183
AmStatus	page 184
AmString	page 185

Exception classes

AmException	page 187
AmErrorException	page 188
AmWarningException	page 189

AmSessionFactory

The **AmSessionFactory** class is used to create AmSession objects.

AmSessionFactory

Constructors for an AmSessionFactory.

```
AmSessionFactory();
AmSessionFactory(char * name);
```

`name`	The name of the AmSessionFactory. This is the location of the data files used by the AMI (the repository file and the local host file). The name should be a fully qualified directory that includes the path under which the files are located. Otherwise, see “Local host and repository files” on page 272 for the location of these files.

createSession

Creates an AmSession object.

```
AmSession * createSession(char * name);
```

`name`	The name of the AmSession.

deleteSession

Deletes an AmSession object previuosly created using the **createSession** method.

```
void  deleteSession(AmSession ** pSession);
```

`pSession`	A pointer to the AmSession pointer returned by the **createSession** method.

getFactoryName

Returns the name of the AmSessionFactory.

```
AmString  getFactoryName();
```

getLocalHost

Returns the name of the local host file.

```
AmString  getLocalHost();
```

getRepository

Returns the name of the repository file.

```
AmString  getRepository();
```

getTraceLevel

Returns the trace level for the AmSessionFactory.

```
int  getTraceLevel();
```

getTraceLocation

Returns the location of the trace for the AmSessionFactory.

```
AmString  getTraceLocation();
```

setLocalHost

Sets the name of the AMI local host file to be used by any AmSession created from this AmSessionFactory. (Otherwise, the default host file `amthost.xml` is used.)

```
void  setLocalHost(char * fileName);
```

`fileName`	The name of the file used by the AMI as the local host file. This file must be present on the local file system or an error will be produced upon the creation of an AmSession.

setRepository

Sets the name of the AMI repository to be used by any AmSession created from this AmSessionFactory. (Otherwise, the default repository file `amt.xml` is used.)

```
void  setRepository(char * fileName);
```

`fileName`	The name of the file used by the AMI as the repository. This file must be present on the local file system or an error will be produced upon the creation of an AmSession.

setTraceLevel

Sets the trace level for the AmSessionFactory.

```
void  setTraceLevel(int level);
```

`level`	The trace level to be set in the AmSessionFactory. Trace levels are 0 through 9, where 0 represents minimal tracing and 9 represents a fully detailed trace.

setTraceLocation

Sets the location of the trace for the AmSessionFactory.

```
void  setTraceLocation(char * location);
```

`location`	The location on the local system where trace files will be written. This location must be a directory, and it must exist prior to the trace being run.

AmSession

An **AmSession** object provides the scope for a unit of work and creates and manages all other objects, including at least one connection object. Each (MQSeries) connection object encapsulates a single MQSeries queue manager connection. The session object definition specifying the required set of queue manager connection(s) can be provided by a repository policy definition, or by default will name a single local queue manager with no repository. The session, when deleted, is responsible for releasing memory by closing and deleting all other objects that it manages.

Note that you should not mix MQSeries MQCONN or MQDISC requests (or their equivalent in the MQSeries C++ interface) on the same thread as AMI calls, otherwise premature disconnection might occur.

begin

Begins a unit of work in this AmSession, allowing an AMI application to take advantage of the resource coordination provided in MQSeries version 5. The unit of work can subsequently be committed by the **commit** method, or backed out by the **rollback** method. This should be used only when AMI is the transaction coordinator. If available, native coordination APIs (for example CICS or Tuxedo) should be used.

begin is overloaded. The `policy` parameter is optional.

```
  void  begin(AmPolicy &policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

clearErrorCodes

Clears the error codes in the AmSession.

```
  void  clearErrorCodes();
```

close

Closes the AmSession, and all open objects owned by it. **close** is overloaded: the `policy` parameter is optional.

```
  void  close(AmPolicy &policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

commit

Commits a unit of work that was started by **AmSession.begin**. **commit** is overloaded: the `policy` parameter is optional.

```
  void  commit(AmPolicy &policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

createDistributionList

Creates an AmDistributionList object.

```
AmDistributionList * createDistributionList(char * name);
```

`name`	The name of the AmDistributionList. This must match the name of a distribution list defined in the repository.

createMessage

Creates an AmMessage object.

```
AmMessage * createMessage(char * name);
```

`name`	The name of the AmMessage. This can be any name that is meaningful to the application.

createPolicy

Creates an AmPolicy object.

```
AmPolicy * createPolicy(char * name);
```

`name`	The name of the AmPolicy. If it matches a policy defined in the repository, the policy will be created using the repository definition, otherwise it will be created with default values.

createPublisher

Creates an AmPublisher object.

```
AmPublisher * createPublisher(char * name);
```

`name`	The name of the AmPublisher. If it matches a publisher defined in the repository, the publisher will be created using the repository definition, otherwise it will be created with default values (that is, with an AmSender name that matches the publisher name).

createReceiver

Creates an AmReceiver object.

```
AmReceiver * createReceiver(char * name);
```

`name`	The name of the AmReceiver. If it matches a receiver defined in the repository, the receiver will be created using the repository definition, otherwise it will be created with default values (that is, with a queue name that matches the receiver name).

createSender

Creates an AmSender object.

```
AmSender * createSender(char * name);
```

`name`	The name of the AmSender. If it matches a sender defined in the repository, the sender will be created using the repository definition, otherwise it will be created with default values (that is, with a queue name that matches the sender name).

createSubscriber

Creates an AmSubscriber object.

```
AmSubscriber * createSubscriber(char * name);
```

`name`	The name of the AmSubscriber. If it matches a subscriber defined in the repository, the subscriber will be created using the repository definition, otherwise it will be created with default values (that is, with an AmSender name that matches the subscriber name, and an AmReceiver name that is the same with the addition of the suffix '.RECEIVER').

deleteDistributionList

Deletes an AmDistributionList object.

```
void  deleteDistributionList(AmDistributionList ** dList);
```

`dList`	A pointer to the AmDistributionList * returned on a createDistributionList call.

deleteMessage

Deletes an AmMessage object.

```
void  deleteMessage(AmMessage ** message);
```

`message`	A pointer to the AmMessage * returned on a createMessage call.

deletePolicy

Deletes an AmPolicy object.

```
void  deletePolicy(AmPolicy ** policy);
```

`policy`	A pointer to the AmPolicy * returned on a createPolicy call.

deletePublisher

Deletes an AmPublisher object.

```
void  deletePublisher(AmPublisher ** publisher);
```

`publisher`	A pointer to the AmPublisher returned on a createPublisher call.

deleteReceiver

Deletes an AmReceiver object.

```
void  deleteReceiver(AmReceiver ** receiver);
```

`receiver`	A pointer to the AmReceiver returned on a createReceiver call.

deleteSender

Deletes an AmSender object.

```
void  deleteSender(AmSender ** sender);
```

`sender`	A pointer to the AmSender returned on a createSender call.

deleteSubscriber

Deletes an AmSubscriber object.

```
void  deleteSubscriber(AmSubscriber ** subscriber);
```

`subscriber`	A pointer to the AmSubscriber returned on a createSubscriber call.

enableWarnings

Enables AmWarningExceptions; the default behavior for any AmObject is that AmWarningExceptions are not raised. Note that warning reason codes can be retrieved using **getLastErrorStatus**, even if AmWarningExceptions are disabled.

```
void  enableWarnings(AMBOOL warningsOn);
```

`warningsOn`	If set to AMB_TRUE, AmWarningExceptions will be raised for this object.

getLastErrorStatus

Returns the AmStatus of the last error condition.

```
AmStatus  getLastErrorStatus();
```

getName

Returns the name of the AmSession.

```
String  getName();
```

getTraceLevel

Returns the trace level of the AmSession.

```
int  getTraceLevel();
```

getTraceLocation

Returns the location of the trace for the AmSession.

```
AmString  getTraceLocation();
```

open

Opens an AmSession using the specified policy. The application profile group of this policy provides the connection definitions enabling the connection objects to be created. The specified library is loaded for each connection and its dispatch table initialized. If the transport type is MQSeries and the MQSeries local queue manager library cannot be loaded, then the MQSeries client queue manager is loaded. Each connection object is then opened.

open is overloaded: the `policy` parameter is optional.

```
void  open(AmPolicy &policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

rollback

Rolls back a unit of work that was started by **AmSession.begin**, or under policy control. **rollback** is overloaded: the `policy` parameter is optional.

```
void  rollback(AmPolicy &policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

AmMessage

An **AmMessage** object encapsulates the MQSeries MQMD message properties, and name/value elements such as the topics for publish/subscribe messages. In addition it contains the application data.

The initial state of the message object is:

`CCSID`	default queue manager CCSID
`correlationId`	all zeroes
`dataLength`	zero
`dataOffset`	zero
`elementCount`	zero
`encoding`	AMENC_NATIVE
`format`	AMFMT_STRING
`groupStatus`	AMGRP_MSG_NOT_IN_GROUP
`topicCount`	zero

When a message object is used to send a message, it might not be left in the same state as it was prior to the send. Therefore, if you use the message object for repeated send operations, it is advisable to reset it to its initial state (see **reset** on page 164) and rebuild it each time.

addElement

Adds a name/value element to an AmMessage object. **addElement** is overloaded: the `element` parameter is required, but the `options` parameter is optional.

```
void  addElement(
  AmElement  &element,
  int        options);
```

`element`	The element to be added to the AmMessage.
`options`	The options to be used. This parameter is reserved and must be set to zero.

addTopic

Adds a publish/subscribe topic to an AmMessage object.

```
void  addTopic(char * topicName);
```

`topicName`	The name of the topic to be added to the AmMessage.

clearErrorCodes

Clears the error in the AmMessage object.

```
void  clearErrorCodes();
```

deleteElement

Deletes the element in the AmMessage object at the specified index. Indexing is within all elements of a message, and might include topics (which are specialized elements).

```
void  deleteElement(int index);
```

`index`	The index of the element to be deleted, starting from zero. On completion, elements with higher `index` values than that specified will have those values reduced by one. **getElementCount** gets the number of elements in the message.

deleteNamedElement

Deletes the element with the specified name in the AmMessage object, at the specified index. Indexing is within all elements that share the same name.

```
void  deleteNamedElement(
  char * name,
  int    index);
```

`name`	The name of the element to be deleted.
`index`	The index of the element to be deleted, starting from zero. On completion, elements with higher `index` values than that specified will have those values reduced by one. **getNamedElementCount** gets the number of elements in the message with the specified name.

deleteTopic

Deletes a publish/subscribe topic in an AmMessage object at the specified index. Indexing is within all topics in the message.

```
void  deleteTopic(int index);
```

`index`	The index of the topic to be deleted, starting from zero. **getTopicCount** gets the number of topics in the message.

enableWarnings

Enables AmWarningExceptions; the default behavior for any AmObject is that AmWarningExceptions are not raised. Note that warning reason codes can be retrieved using **getLastErrorStatus**, even if AmWarningExceptions are disabled.

```
void  enableWarnings(AMBOOL warningsOn);
```

`warningsOn`	If set to AMB_TRUE, AmWarningExceptions will be raised for this object.

getCCSID

Returns the coded character set identifier used by the AmMessage.

```
int  getCCSID();
```

getCorrelationId

Returns the correlation identifier for the AmMessage.

```
AmBytes  getCorrelationId();
```

getDataLength

Returns the length of the message data in the AmMessage.

```
int  getDataLength();
```

getDataOffset

Returns the current offset in the message data for reading or writing data bytes.

```
int  getDataOffset();
```

getElement

Returns an element in an AmMessage object at the specified index. Indexing is within all elements in the message, and might include topics (which are specialized elements).

```
AmElement  getElement(int index);
```

`index`	The index of the element to be returned, starting from zero. **getElementCount** gets the number of elements in the message.

getElementCount

Returns the total number of elements in an AmMessage object. This might include topics (which are specialized elements).

```
int  getElementCount();
```

getEncoding

Returns the value used to encode numeric data types for the AmMessage.

```
int  getEncoding();
```

The following values can be returned:

```
AMENC_NATIVE
AMENC_NORMAL
AMENC_NORMAL_FLOAT_390
AMENC_REVERSED
AMENC_REVERSED_FLOAT_390
AMENC_UNDEFINED
```

getFormat

Returns the format of the AmMessage.

```
AmString  getFormat();
```

The following values can be returned:

```
AMFMT_NONE
AMFMT_STRING
AMFMT_RF_HEADER
```

getGroupStatus

Returns the group status value for the AmMessage. This indicates whether the message is in a group, and if it is the first, middle, last or only one in the group.

```
int  getGroupStatus();
```

The following values can be returned:

```
AMGRP_MSG_NOT_IN_GROUP
AMGRP_FIRST_MSG_IN_GROUP
AMGRP_MIDDLE_MSG_IN_GROUP
AMGRP_LAST_MSG_IN_GROUP
AMGRP_ONLY_MSG_IN_GROUP
```

Alternatively, bitwise tests can be performed using the constants:

```
AMGF_IN_GROUP
AMGF_FIRST
AMGF_LAST
```

getLastErrorStatus

Returns the AmStatus of the last error condition for this object.

```
AmStatus  getLastErrorStatus();
```

getMessageId

Returns the message identifier from the AmMessage object.

```
AmBytes  getMessageId();
```

getName

Returns the name of the AmMessage object.

```
AmString  getName();
```

getNamedElement

Returns the element with the specified name in an AmMessage object, at the specified index. Indexing is within all elements that share the same name.

```
AmElement  getNamedElement(
  char *  name,
  int     index);
```

`name`	The name of the element to be returned.
`index`	The index of the element to be returned, starting from zero.

getNamedElementCount

Returns the total number of elements with the specified name in the AmMessage object.

```
int  getNamedElementCount(char * name);
```

`name`	The name of the elements to be counted.

getTopic

Returns the publish/subscribe topic in the AmMessage object, at the specified index. Indexing is within all topics.

```
AmString  getTopic(int index);
```

`index`	The index of the topic to be returned, starting from zero. **getTopicCount** gets the number of topics in the message.

getTopicCount

Returns the total number of publish/subscribe topics in the AmMessage object.

```
int  getTopicCount();
```

readBytes

Populates an AmByte object with data from the AmMessage, starting at the current data offset (which must be positioned before the end of the data for the read to be successful). Use **setDataOffset** to specify the data offset. **readBytes** will advance the data offset by the number of bytes read, leaving the offset immediately after the last byte read.

```
AmBytes  readBytes(int dataLength);
```

`dataLength`	The maximum number of bytes to be read from the message data. The number of bytes returned is the minimum of `dataLength` and the number of bytes between the data offset and the end of the data.

reset

Resets the AmMessage object to its initial state (see page 159).

reset is overloaded: the `options` parameter is optional.

```
void  reset(int options);
```

`options`	A reserved field that must be set to zero.

setCCSID

Sets the coded character set identifier used by the AmMessage object.

```
void  setCCSID(int codedCharSetId);
```

`codedCharSetId`	The CCSID to be set in the AmMessage.

setCorrelationId

Sets the correlation identifier in the AmMessage object.

```
void  setCorrelationId(AmBytes &correlId);
```

`correlId`	An AmBytes object containing the correlation identifier to be set in the AmMessage.

setDataOffset

Sets the data offset for reading or writing byte data.

```
void  setDataOffset(int dataOffset);
```

`dataOffset`	The data offset to be set in the AmMessage. Set an offset of zero to read or write from the start of the data.

setEncoding

Sets the encoding of the data in the AmMessage object.

```
void  setEncoding(int encoding);
```

`encoding` The encoding to be used in the AmMessage. It can take one of the following values:

```
AMENC_NATIVE
AMENC_NORMAL
AMENC_NORMAL_FLOAT_390
AMENC_REVERSED
AMENC_REVERSED_FLOAT_390
AMENC_UNDEFINED
```

setFormat

Sets the format for the AmMessage object.

```
  void  setFormat(char * format);
```

`format`	The format to be used in the AmMessage. It can take one of the following values:
	`AMFMT_NONE`
	`AMFMT_STRING`
	`AMFMT_RF_HEADER`
	If set to AMFMT_NONE, the default format for the sender will be used (if available).

setGroupStatus

Sets the group status value for the AmMessage. This indicates whether the message is in a group, and if it is the first, middle, last or only one in the group. Once you start sending messages in a group, you must complete the group before sending any messages that are not in the group.

If you specify AMGRP_MIDDLE_MSG_IN_GROUP or AMGRP_LAST_MSG_IN_GROUP without specifying AMGRP_FIRST_MSG_IN_GROUP, the behavior is the same as for AMGRP_FIRST_MSG_IN_GROUP and AMGRP_ONLY_MSG_IN_GROUP.

If you specify AMGRP_FIRST_MSG_IN_GROUP out of sequence, then the behavior is the same as for AMGRP_MIDDLE_MSG_IN_GROUP.

```
  void  setGroupStatus(int groupStatus);
```

`groupStatus`	The group status to be set in the AmMessage. It can take one of the following values:
	`AMGRP_MSG_NOT_IN_GROUP`
	`AMGRP_FIRST_MSG_IN_GROUP`
	`AMGRP_MIDDLE_MSG_IN_GROUP`
	`AMGRP_LAST_MSG_IN_GROUP`
	`AMGRP_ONLY_MSG_IN_GROUP`

writeBytes

Writes a byte array into the AmMessage object, starting at the current data offset. If the data offset is not at the end of the data, existing data is overwritten. Use **setDataOffset** to specify the data offset. **writeBytes** will advance the data offset by the number of bytes written, leaving it immediately after the last byte written.

```
  void  writeBytes(AmBytes &data);
```

`data`	An AmBytes object containing the data to be written to the AmMessage.

AmSender

An **AmSender** object encapsulates an MQSeries object descriptor (MQOD) structure. This represents an MQSeries queue on a local or remote queue manager. An open sender service is always associated with an open connection object (such as a queue manager connection). Support is also included for dynamic sender services (those that encapsulate model queues). The required sender service object definitions can be provided from a repository, or created without a repository definition by defaulting to the existing queue objects on the local queue manager.

The AmSender object must be created before it can be opened. This is done using **AmSession.createSender**.

A *responder* is a special type of AmSender used for sending a response to a request message. It is not created from a repository definition. Once created, it must not be opened until used in its correct context as a responder receiving a request message with **AmReceiver.receive**. When opened, its queue and queue manager properties are modified to reflect the `ReplyTo` destination specified in the message being received. When first used in this context, the sender service becomes a responder sender service.

clearErrorCodes

Clears the error codes in the AmSender.

```
  void  clearErrorCodes();
```

close

Closes the AmSender. **close** is overloaded: the `policy` parameter is optional.

```
  void  close(AmPolicy &policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

enableWarnings

Enables AmWarningExceptions; the default behavior for any AmObject is that AmWarningExceptions are not raised. Note that warning reason codes can be retrieved using **getLastErrorStatus**, even if AmWarningExceptions are disabled.

```
  void  enableWarnings(AMBOOL warningsOn);
```

`warningsOn`	If set to AMB_TRUE, AmWarningExceptions will be raised for this object.

getCCSID

Returns the coded character set identifier for the AmSender. A non-default value reflects the CCSID of a remote system unable to perform CCSID conversion of received messages. In this case the sender must perform CCSID conversion of the message before it is sent.

```
  int  getCCSID();
```

getEncoding

Returns the value used to encode numeric data types for the AmSender. A non-default value reflects the encoding of a remote system unable to convert the encoding of received messages. In this case the sender must convert the encoding of the message before it is sent.

```
int  getEncoding();
```

getLastErrorStatus

Returns the AmStatus of the last error condition.

```
AmStatus  getLastErrorStatus();
```

getName

Returns the name of the AmSender.

```
AmString  getName();
```

open

Opens an AmSender service. **open** is overloaded: the `policy` parameter is optional.

```
void  open(AmPolicy &policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

send

Sends a message using the AmSender service. If the AmSender is not open, it will be opened (if this action is specified in the policy options).

send is overloaded: the `sendMessage` parameter is required, but the others are optional. `receivedMessage` and `responseService` are used in request/response messaging, and are mutually exclusive.

```
void  send(
  AmMessage   &sendMessage,
  AmReceiver  &responseService,
  AmMessage   &receivedMessage,
  AmPolicy    &policy);
```

`sendMessage`	The message object that contains the data to be sent.
`responseService`	The AmReceiver to which the response to this message should be sent. Omit it if no response is required.
`receivedMessage`	The previously received message which is used for correlation with the sent message. If omitted, the sent message is not correlated with any received message.
`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

AmReceiver

An **AmReceiver** object encapsulates an MQSeries object descriptor (MQOD) structure. This represents an MQSeries queue on a local or remote queue manager. An open AmReceiver is always associated with an open connection object, such as a queue manager connection. Support is also included for a dynamic AmReceiver (that encapsulates a model queue). The required AmReceiver object definitions can be provided from a repository or can be created automatically from the set of existing queue objects available on the local queue manager.

There is a definition type associated with each AmReceiver:

```
AMDT_UNDEFINED
AMDT_TEMP_DYNAMIC
AMDT_DYNAMIC
AMDT_PREDEFINED
```

An AmReceiver created from a repository definition will be initially of type AMDT_PREDEFINED or AMDT_DYNAMIC. When opened, its definition type might change from AMDT_DYNAMIC to AMDT_TEMP_DYNAMIC according to the properties of its underlying queue object.

An AmReceiver created with default values (that is, without a repository definition) will have its definition type set to AMDT_UNDEFINED until it is opened. When opened, this will become AMDT_DYNAMIC, AMDT_TEMP_DYNAMIC, or AMDT_PREDEFINED, according to the properties of its underlying queue object.

browse

Browses an AmReceiver service. **browse** is overloaded: the `browseMessage` and `options` parameters are required, but the others are optional.

```
void  browse(
  AmMessage   &browseMessage,
  int         options,
  AmSender    &responseService,
  AmPolicy    &policy);
```

`browseMessage` The message object that receives the browse data.

`options` Options controlling the browse operation. Possible values are:

```
AMBRW_NEXT
AMBRW_FIRST
AMBRW_CURRENT
AMBRW_RECEIVE_CURRENT
AMBRW_DEFAULT           (AMBRW_NEXT)
AMBRW_LOCK_NEXT         (AMBRW_LOCK + AMBRW_NEXT)
AMBRW_LOCK_FIRST        (AMBRW_LOCK + AMBRW_FIRST)
AMBRW_LOCK_CURRENT      (AMBRW_LOCK + AMBRW_CURRENT)
AMBRW_UNLOCK
```

`AMBRW_RECEIVE_CURRENT` is equivalent to **AmReceiver.receive** for the message under the browse cursor.

Note that a locked message is unlocked by another browse or receive, even though it is not for the same message.

`responseService`	The AmSender to be used for sending any response to the browsed message. If omitted, no response can be sent.
`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

clearErrorCodes

Clears the error codes in the AmReceiver.

```
  void  clearErrorCodes();
```

close

Closes the AmReceiver. **close** is overloaded: the `policy` parameter is optional.

```
  void  close(AmPolicy &policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

enableWarnings

Enables AmWarningExceptions; the default behavior for any AmObject is that AmWarningExceptions are not raised. Note that warning reason codes can be retrieved using **getLastErrorStatus**, even if AmWarningExceptions are disabled.

```
  void  enableWarnings(AMBOOL warningsOn);
```

`warningsOn`	If set to AMB_TRUE, AmWarningExceptions will be raised for this object.

getDefinitionType

Returns the definition type (service type) for the AmReceiver.

```
  int  getDefinitionType();
```

The following values can be returned:

```
AMDT_UNDEFINED
AMDT_TEMP_DYNAMIC
AMDT_DYNAMIC
AMDT_PREDEFINED
```

Values other than AMDT_UNDEFINED reflect the properties of the underlying queue object.

getLastErrorStatus

Returns the AmStatus of the last error condition.

```
  AmStatus  getLastErrorStatus();
```

getName

Returns the name of the AmReceiver.

```
AmString  getName();
```

getQueueName

Returns the queue name of the AmReceiver. This is used to determine the queue name of a permanent dynamic AmReceiver, so that it can be recreated with the same queue name in order to receive messages in a subsequent session. (See also **setQueueName**.)

```
AmString  getQueueName();
```

open

Opens an AmReceiver service. **open** is overloaded: the `policy` parameter is optional.

```
void  open(AmPolicy &policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

receive

Receives a message from the AmReceiver service. **receive** is overloaded: the `receiveMessage` parameter is required, but the others are optional.

```
void  receive(
  AmMessage   &receiveMessage,
  AmSender    &responseService,
  AmMessage   &selectionMessage,
  AmPolicy    &policy);
```

`receiveMessage`	The message object that receives the data. The message object is reset implicitly before the receive takes place.
`responseService`	The AmSender to be used for sending any response to the received message. If omitted, no response can be sent.
`selectionMessage`	A message object which contains the correlation ID used to selectively receive a message from the AmReceiver. If omitted, the first available message is received.
`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

setQueueName

Sets the queue name of the AmReceiver (when this encapsulates a model queue). This is used to specify the queue name of a recreated permanent dynamic AmReceiver, in order to receive messages in a session subsequent to the one in which it was created. (See also **getQueueName**.)

```
void  setQueueName(char * queueName);
```

`queueName`	The queue name to be set in the AmReceiver.

AmDistributionList

An **AmDistributionList** object encapsulates a list of AmSender objects.

clearErrorCodes

Clears the error codes in the AmDistributionList.

```
void  clearErrorCodes();
```

close

Closes the AmDistributionList. **close** is overloaded: the `policy` parameter is optional.

```
void  close(AmPolicy &policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

enableWarnings

Enables AmWarningExceptions; the default behavior for any AmObject is that AmWarningExceptions are not raised. Note that warning reason codes can be retrieved using **getLastErrorStatus**, even if AmWarningExceptions are disabled.

```
void  enableWarnings(AMBOOL warningsOn);
```

`warningsOn`	If set to AMB_TRUE, AmWarningExceptions will be raised for this object.

getLastErrorStatus

Returns the AmStatus of the last error condition of this object.

```
AmStatus  getLastErrorStatus();
```

getName

Returns the name of the AmDistributionList object.

```
AmString  getName();
```

getSender

Returns a pointer to the AmSender object contained within the AmDistributionList object at the index specified. `AmDistributionList.getSenderCount` gets the number of AmSender services in the distribution list.

```
AmSender * getSender(int index);
```

`index`	The index of the AmSender in the AmDistributionList, starting at zero.

getSenderCount

Returns the number of AmSender services in the AmDistributionList object.

```
  int  getSenderCount();
```

open

Opens an AmDistributionList object for each of the destinations in the distribution list. **open** is overloaded: the `policy` parameter is optional.

```
  void  open(AmPolicy &policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

send

Sends a message to each AmSender defined in the AmDistributionList object. **send** is overloaded: the `sendMessage` parameter is required, but the others are optional.

```
  void  send(
    AmMessage   &sendMessage,
    AmReceiver  &responseService,
    AmPolicy    &policy);
```

`sendMessage`	The message object containing the data to be sent.
`responseService`	The AmReceiver to be used for receiving any response to the sent message. If omitted, no response can be received.
`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

AmPublisher

An **AmPublisher** object encapsulates an AmSender and provides support for publish requests to a publish/subscribe broker.

clearErrorCodes

Clears the error codes in the AmPublisher.

```
void  clearErrorCodes();
```

close

Closes the AmPublisher. **close** is overloaded: the `policy` parameter is optional.

```
void  close(AmPolicy &policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

enableWarnings

Enables AmWarningExceptions; the default behavior for any AmObject is that AmWarningExceptions are not raised. Note that warning reason codes can be retrieved using **getLastErrorStatus**, even if AmWarningExceptions are disabled.

```
void  enableWarnings(AMBOOL warningsOn);
```

`warningsOn`	If set to AMB_TRUE, AmWarningExceptions will be raised for this object.

getCCSID

Returns the coded character set identifier for the AmPublisher. A non-default value reflects the CCSID of a remote system unable to perform CCSID conversion of received messages. In this case the publisher must perform CCSID conversion of the message before it is sent.

```
int  getCCSID();
```

getEncoding

Returns the value used to encode numeric data types for the AmPublisher. A non-default value reflects the encoding of a remote system unable to convert the encoding of received messages. In this case the publisher must convert the encoding of the message before it is sent.

```
int  getEncoding();
```

getLastErrorStatus

Returns the AmStatus of the last error condition.

```
AmStatus  getLastErrorStatus();
```

getName

Returns the name of the AmPublisher.

```
AmString  getName();
```

open

Opens an AmPublisher service. **open** is overloaded: the `policy` parameter is optional.

```
void  open(AmPolicy &policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

publish

Publishes a message using the AmPublisher. **publish** is overloaded: the `pubMessage` parameter is required, but the others are optional.

```
void  publish(
  AmMessage   &pubMessage,
  AmReceiver  &responseService,
  AmPolicy    &policy);
```

`pubMessage`	The message object that contains the data to be published.
`responseService`	The AmReceiver to which the response to this publish request should be sent. Omit it if no response is required. This parameter is mandatory if the policy specifies implicit registration of the publisher.
`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

AmSubscriber

An **AmSubscriber** object encapsulates both an AmSender and an AmReceiver. It provides support for subscribe and unsubscribe requests to a publish/subscribe broker, and for receiving publications from the broker.

clearErrorCodes

Clears the error codes in the AmSubscriber.

```
void  clearErrorCodes();
```

close

Closes the AmSubscriber. **close** is overloaded: the `policy` parameter is optional.

```
void  close(AmPolicy &policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

enableWarnings

Enables AmWarningExceptions; the default behavior for any AmObject is that AmWarningExceptions are not raised. Note that warning reason codes can be retrieved using **getLastErrorStatus**, even if AmWarningExceptions are disabled.

```
void  enableWarnings(AMBOOL warningsOn);
```

`warningsOn`	If set to AMB_TRUE, AmWarningExceptions will be raised for this object.

getCCSID

Returns the coded character set identifier for the AmSender in the AmSubscriber. A non-default value reflects the CCSID of a remote system unable to perform CCSID conversion of received messages. In this case the subscriber must perform CCSID conversion of the message before it is sent.

```
int  getCCSID();
```

getDefinitionType

Returns the definition type for the AmReceiver in the AmSubscriber.

```
int  getDefinitionType();
```

The following values can be returned:

```
AMDT_UNDEFINED
AMDT_TEMP_DYNAMIC
AMDT_DYNAMIC
AMDT_PREDEFINED
```

getEncoding

Returns the value used to encode numeric data types for the AmSender in the AmSubscriber. A non-default value reflects the encoding of a remote system unable to convert the encoding of received messages. In this case the subscriber must convert the encoding of the message before it is sent.

```
int  getEncoding();
```

getLastErrorStatus

Returns the AmStatus of the last error condition.

```
AmStatus  getLastErrorStatus();
```

getName

Returns the name of the AmSubscriber.

```
AmString  getName();
```

getQueueName

Returns the queue name used by the AmSubscriber to receive messages. This is used to determine the queue name of a permanent dynamic AmReceiver in the AmSubscriber, so that it can be recreated with the same queue name in order to receive messages in a subsequent session. (See also **setQueueName**.)

```
AmString  getQueueName();
```

open

Opens an AmSubscriber. **open** is overloaded: the `policy` parameter is optional.

```
void  open(AmPolicy &policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

receive

Receives a message, normally a publication, using the AmSubscriber. The message data, topic and other elements can be accessed using the message interface methods (see page 159).

receive is overloaded: the `pubMessage` parameter is required, but the others are optional.

```
void  receive(
  AmMessage   &pubMessage,
  AmMessage   &selectionMessage,
  AmPolicy    &policy);
```

`pubMessage`	The message object containing the data that has been published. The message object is reset implicitly before the receive takes place.
`selectionMessage`	A message object containing the correlation ID used to selectively receive a message from the AmSubscriber. If omitted, the first available message is received.
`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

setQueueName

Sets the queue name in the AmReceiver of the AmSubscriber, when this encapsulates a model queue. This is used to specify the queue name of a recreated permanent dynamic AmReceiver, in order to receive messages in a session subsequent to the one in which it was created. (See also **setQueueName**.)

```
void  setQueueName(char * queueName);
```

`queueName`	The queue name to be set.

subscribe

Sends a subscribe message to a publish/subscribe broker using the AmSubscriber, to register a subscription. The topic and other elements can be specified using the message interface methods (see page 159) before sending the message.

Publications matching the subscription are sent to the AmReceiver associated with the AmSubscriber. By default, this has the same name as the AmSubscriber, with the addition of the suffix '.RECEIVER'.

subscribe is overloaded: the `subMessage` parameter is required, but the others are optional.

```
void  subscribe(
  AmMessage   &subMessage,
  AmReceiver  &responseService,
  AmPolicy    &policy);
```

`subMessage`	The message object that contains the topic subscription data.
`responseService`	The AmReceiver to which the response to this subscribe request should be sent. Omit it if no response is required. This is not the AmReceiver to which publications will be sent by the broker; they are sent to the AmReceiver associated with the AmSubscriber (see above).
`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

unsubscribe

Sends an unsubscribe message to a publish/subscribe broker using the AmSubscriber, to deregister a subscription. The topic and other elements can be specified using the message interface methods (see page 159) before sending the message.

unsubscribe is overloaded: the `unsubMessage` parameter is required, but the others are optional.

```
void  unsubscribe(
  AmMessage   &unsubMessage,
  AmReceiver  &responseService,
  AmPolicy    &policy);
```

`unsubMessage`	The message object that contains the topics to which the unsubscribe request applies.
`responseService`	The AmReceiver to which the response to this unsubscribe request should be sent. Omit it if no response is required.
`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

AmPolicy

An **AmPolicy** object encapsulates details of how the AMI processes the message (for instance, the priority and persistence of the message, how errors are handled, and whether transactional processing is used).

clearErrorCodes

Clears the error codes in the AmPolicy.

```
void  clearErrorCodes();
```

enableWarnings

Enables AmWarningExceptions; the default behavior for any AmObject is that AmWarningExceptions are not raised. Note that warning reason codes can be retrieved using **getLastErrorStatus**, even if AmWarningExceptions are disabled.

```
void  enableWarnings(AMBOOL warningsOn);
```

`warningsOn`	If set to AMB_TRUE, AmWarningExceptions will be raised for this object.

getLastErrorStatus

Returns the AmStatus of the last error condition.

```
AmStatus  getLastErrorStatus();
```

getName

Returns the name of the AmPolicy object.

```
AmString  getName();
```

getWaitTime

Returns the wait time (in ms) set for this AmPolicy.

```
int  getWaitTime();
```

setWaitTime

Sets the wait time for any **receive** using this AmPolicy.

```
void  setWaitTime(int waitTime);
```

`waitTime`	The wait time (in ms) to be set in the AmPolicy.

AmBytes

An **AmBytes** object encapsulates an array of bytes. It allows the AMI to pass bytes across the interface and enables manipulation of these bytes.

cmp

Methods used to compare AmBytes objects. These methods return 0 if the data is the same, and 1 otherwise.

```
AMLONG cmp(const AmBytes &amBytes);
AMLONG cmp(const char * stringData);
AMLONG cmp(const char * charData, AMLONG length);
```

`amBytes`	A reference to the AmBytes object being compared.
`stringData`	A char pointer to the NULL terminated string being compared.
`charData`	A char pointer to the bytes being compared.
`length`	The length, in bytes, of the data to be compared. If this length is not the same as the length of the AmBytes object, the comparison fails.

constructors

Constructors for an AmBytes object.

```
AmBytes();
AmBytes(const AmBytes &amBytes);
AmBytes(const AMBYTE byte);
AmBytes(const AMLONG long);
AmBytes(const char * charData);
AmBytes(const AmString &amString);
AmBytes(const AMSTR stringData);
AmBytes(const AMBYTE *character, const AMLONG length);
```

`amBytes`	A reference to an AmBytes object used to create the new AmBytes object.
`byte`	A single byte used to create the new AmBytes object.
`long`	An AMLONG used to create the new AmBytes object.
`charData`	A char pointer to a NULL terminated string used to create the new AmBytes object.
`stringData`	A NULL terminated string used to create the new AmBytes object.
`character`	The character to populate the new AmBytes object with.
`length`	The length, in bytes, of the new AmBytes object.

cpy

Methods used to copy from an AmBytes object. Any existing data in the AmBytes object is discarded.

```
AmBytes &cpy();
AmBytes &cpy(const AMSTR stringData);
AmBytes &cpy(const AMBYTE *byteData, const AMLONG length);
AmBytes &cpy(const AMBYTE byte);
AmBytes &cpy(const AMLONG long);
AmBytes &cpy(const AmBytes &amBytes);
```

`stringData`	A NULL terminated string being copied.
`byteData`	A pointer to the bytes being copied.
`length`	The length, in bytes, of the data to be copied.
`byte`	The single byte being copied.
`long`	An AMLONG being copied.
`amBytes`	A reference to the AmBytes object being copied.

dataPtr

Method to reference the byte data contained within an AmBytes object.

```
const AMBYTE * dataPtr() const;
```

destructor

Destructor for an AmBytes object.

```
~AmBytes();
```

length

Returns the length of an AmBytes object.

```
AMLONG length();
```

operators

Operators for an AmBytes object.

```
AmBytes &operator = (const AmBytes &);
AMBOOL operator == (const AmBytes &) const;
AMBOOL operator != (const AmBytes &) const;
```

pad

Method used to pad AmBytes objects with a specified byte value.

```
AmBytes &pad(const AMLONG length, const AMBYTE byte);
```

`length`	The required length of the AmBytes after the padding.
`byte`	The byte value used to pad the AmBytes object.

AmElement

An **AmElement** object encapsulates a name/value pair which can be added to an AmMessage object.

AmElement

Constructor for an AmElement object.

```
AmElement(char * name, char * value);
```

`name`	The name of the element.
`value`	The value of the element.

getName

Returns the name of the AmElement.

```
AmString  getName();
```

getValue

Returns the value of the AmElement.

```
AmString  getValue();
```

getVersion

Returns the version of the AmElement (the default value is AMELEM_VERSION_1).

```
int  getVersion();
```

setVersion

Sets the version of the AmElement.

```
void  setVersion(int version);
```

`version`	The version of the AmElement that is set. It can take the value AMELEM_VERSION_1 or AMELEM_CURRENT_VERSION.

toString

Returns a AmString representation of the AmElement.

```
AmString  toString();
```

AmObject

AmObject is a virtual class. The following classes inherit from the AmObject class:

AmSession
AmMessage
AmSender
AmDistributionList
AmReceiver
AmPublisher
AmSubscriber
AmPolicy

This allows application programmers to use generic error handling routines.

clearErrorCodes

Clears the error codes in the AmObject.

```
void  clearErrorCodes();
```

getLastErrorStatus

Returns the AmStatus of the last error condition.

```
AmStatus  getLastErrorStatus();
```

getName

Returns the name of the AmObject.

```
AmString  getName();
```

AmStatus

An **AmStatus** object encapsulates the error status of other AmObjects.

AmStatus

Constructor for an AmStatus object.

```
AmStatus();
```

getCompletionCode

Returns the completion code from the AmStatus object.

```
int  getCompletionCode();
```

getReasonCode

Returns the reason code from the AmStatus object.

```
int  getReasonCode();
```

getReasonCode2

Returns the secondary reason code from the AmStatus object. (This code is specific to the underlying transport used by the AMI). For MQSeries, the secondary reason code is an MQSeries reason code of type MQRC_xxx.

```
int  getReasonCode2();
```

toString

Returns an AmString representation of the internal state of the AmStatus object.

```
AmString  toString();
```

AmString

An **AmString** object encapsulates a string or array of characters. It allows the AMI to pass strings across the interface and enables manipulation of these strings.

cat

Methods used to concatenate.

```
AmString &cat(const AmString &amString);
AmString &cat(const AMSTR stringData);
```

`amString`	A reference to the AmString object being concatenated.
`stringData`	The NULL terminated string being concatenated into the AmString object.

cmp

Methods to compare AmStrings with AmStrings and data of type AMSTR. A return value of 0 indicates that the two strings match exactly.

```
AMLONG cmp(const AmString &amString) const;
AMLONG cmp(const AMSTR stringData) const;
```

`amString`	A reference to the AmString object being compared.
`stringData`	The NULL terminated string being compared.

constructors

Constructors for an AmString object.

```
AmString();
AmString(const AmString &amString);
AmString(const AMSTR stringData);
```

`amString`	A reference to an AmString object used to create the new AmString.
`stringData`	A NULL terminated string, from which the AmString is constructed.

contains

Method to indicate whether a specified character is contained within the AmString.

```
AMBOOL contains(const AMBYTE character) const;
```

`character`	The character being used for the search.

cpy

Methods used to copy from an AmString. Any existing data in the AmString is discarded.

```
AmString &cpy(const AmString &amString);
AmString &cpy(const AMSTR stringData);
```

`amString`	A reference to an AmString object being copied.
`stringData`	The NULL terminated string being copied into the AmString.

destructor

Destructor for an AmString object.

```
  ~AmString();
```

operators

Operators for an AmString object.

```
AmString &operator = (const AmString &);
AmString &operator = (const AMSTR);
AMBOOL operator == (const AmString &) const;
AMBOOL operator != (const AmString &) const;
```

pad

Method used to pad AmStrings with a specified character.

```
AmString &pad(const AMLONG length, const AMBYTE character);
```

`length`	The required length of the AmString after the padding.
`charString`	The character used to pad the AmString.

split

Method used to split AmStrings at the first occurrence of a specified character.

```
AmString &split(AmString &newString, const AMBYTE splitCharacter);
```

`newString`	A reference to an AmString object to contain the latter half of the split string.
`splitCharacter`	The first character at which the split will occur.

strip

Method used to strip leading and trailing blanks from AmStrings.

```
AmString &strip();
```

length

Returns the length of an AmString.

```
AMLONG length();
```

text

Method to reference the string contained within an AmString.

```
AMSTR text() const;
```

truncate

Method used to truncate AmStrings.

```
AmString &truncate(const AMLONG length);
```

`length`	The length to which the AmString is to be truncated.

AmException

AmException is the base Exception class; all other Exceptions inherit from this class.

getClassName

Returns the type of object throwing the Exception.

```
AmString  getClassName();
```

getCompletionCode

Returns the completion code for the Exception.

```
int  getCompletionCode();
```

getMethodName

Returns the name of the method throwing the Exception.

```
AmString  getMethodName();
```

getReasonCode

Returns the reason code for the Exception.

```
int  getReasonCode();
```

getSource

Returns the AmObject throwing the Exception.

```
AmObject  getSource();
```

toString

Returns an AmString representation of the Exception.

```
AmString  toString();
```

AmErrorException

An Exception of type **AmErrorException** is raised when an object experiences an error with a severity level of FAILED (CompletionCode = AMCC_FAILED).

getClassName

Returns the type of object throwing the Exception.

```
AmString  getClassName();
```

getCompletionCode

Returns the completion code for the Exception.

```
int  getCompletionCode();
```

getMethodName

Returns the name of the method throwing the Exception.

```
AmString  getMethodName();
```

getReasonCode

Returns the reason code for the Exception.

```
int  getReasonCode();
```

getSource

Returns the AmObject throwing the Exception.

```
AmObject  getSource();
```

toString

Returns an AmString representation of the Exception.

```
AmString  toString();
```

AmWarningException

An Exception of type **AmWarningException** is raised when an object experiences an error with a severity level of WARNING (CompletionCode = AMCC_WARNING).

getClassName

Returns the type of object throwing the Exception.

```
AmString  getClassName();
```

getCompletionCode

Returns the completion code for the Exception.

```
int  getCompletionCode();
```

getMethodName

Returns the name of the method throwing the Exception.

```
AmString  getMethodName();
```

getReasonCode

Returns the reason code for the Exception.

```
int  getReasonCode();
```

getSource

Returns the AmObject throwing the Exception.

```
AmObject  getSource();
```

toString

Returns an AmString representation of the Exception.

```
AmString  toString();
```

Part 4. The Java interface

This part contains:

Chapter 9. Using the Application Messaging Interface in Java

The Application Messaging Interface for Java (amJava) provides a Java style of programming, while being consistent with the object-style interface of the Application Messaging Interface for C. It uses a Java Native Interface (JNI) library, so it cannot be used to write Applets to run in a browser environment.

This chapter describes the following:

- “Structure of the AMI”
- “Writing applications in Java” on page 195
- “Building Java applications” on page 204

Note that the term *object* is used in this book in the object-oriented programming sense, not in the sense of MQSeries ‘objects’ such as channels and queues.

Structure of the AMI

The following classes are provided:

Base classes

AmSessionFactory	Creates AmSession objects.
AmSession	Creates objects within the AMI session, and controls transactional support.
AmMessage	Contains the message data, message ID and correlation ID, and options that are used when sending or receiving a message (most of which come from the policy definition).
AmSender	This is a service that represents a destination (such as an MQSeries queue) to which messages are sent.
AmReceiver	This is a service that represents a source (such as an MQSeries queue) from which messages are received.
AmDistributionList	Contains a list of sender services to provide a list of destinations.
AmPublisher	Contains a sender service where the destination is a publish/subscribe broker.
AmSubscriber	Contains a sender service (to send subscribe and unsubscribe messages to a publish/subscribe broker) and a receiver service (to receive publications from the broker).
AmPolicy	Defines how the message should be handled, including items such as priority, persistence, and whether it is included in a unit of work.

Interface and helper classes

AmObject	This is a Java interface, which is implemented by the base classes listed above (with the exception of AmSessionFactory).
AmConstants	This encapsulates all of the constants needed by amJava.
AmElement	This encapsulates name/value pairs that can be added to AmMessage objects.
AmStatus	This encapsulates the error status of amJava objects.

Exception classes

AmException	This is the base Exception class for amJava; all other amJava Exceptions inherit from this class.
AmErrorException	An Exception of this type is raised when an amJava object experiences an error with a severity level of FAILED (CompletionCode = AMCC_FAILED).
AmWarningException	An Exception of this type is raised when an amJava object experiences an error with a severity level of WARNING (CompletionCode = AMCC_WARNING), provided that warnings have been enabled using the **enableWarnings** method.

Using the repository

You can run AMI applications with or without a repository. If you don't have a repository, you can create an object by specifying its name in a method. It will be created using the appropriate system provided definition (see "System provided definitions" on page 280).

If you have a repository, and you specify the name of an object in a method that matches a name in the repository, the object will be created using the repository definition. (If no matching name is found in the repository, the system provided definition will be used.)

System default objects

The set of system default objects created in C is not accessible directly in Java, but the SYSTEM.DEFAULT.POLICY (constant: AMSD_POL) is used to provide default behavior when a policy is not specified. Objects with identical properties to the system default objects can be created for use in Java using the built-in definitions (see "System provided definitions" on page 280).

Writing applications in Java

This section gives a number of examples showing how to access the Application Messaging Interface using Java.

Many of the method calls are overloaded and in some cases this results in default objects being used. One example of this is the AmPolicy object which can be passed on many of the methods. For example:

Method overloading

```
mySender.send(mySendMessage, myPolicy);

mySender.send(mySendMessage);
```

If a policy has been created to provide specific send behavior, use the first example. However, if the default policy is acceptable, use the second example.

The defaulting of behavior using method overloading is used throughout the examples.

Creating and opening objects

Before using the AMI, you must create and open the required objects. Objects are created with names, which might correspond to named objects in the repository. In the case of the creation of a response sender (`myResponder`) in the example below, the default name for a response type object is specified using the **AmConstants** helper class, so the object is created with default responder values.

Creating AMI objects

```
mySessionFactory = new AmSessionFactory("MY.SESSION.FACTORY");
mySession = mySessionFactory.createSession("MY.SESSION");
myPolicy = mySession.createPolicy("MY.POLICY");

mySender = mySession.createSender("AMT.SENDER.QUEUE");
myReceiver = mySession.createReceiver("AMT.RECEIVER.QUEUE");
myResponder = mySession.createSender(AmConstants.AMDEF_RSP_SND);

mySendMessage = mySession.createMessage("MY.SEND.MESSAGE");
myReceiveMessage = mySession.createMessage("MY.RECEIVE.MESSAGE");
```

The objects are then opened. In the following examples, the session object is opened with the default policy, whereas the sender and receiver objects are opened with a specified policy (`myPolicy`).

Opening the AMI objects

```
mySession.open();
mySender.open(myPolicy);
myReceiver.open(myPolicy);
```

Sending messages

The examples in this section show how to send a datagram (send and forget) message. First, the message data is written to the `mySendMessage` object. Data is always sent in byte form, so the Java **getBytes** method is used to extract the String data as bytes prior to adding to the message.

Writing data to a message object

```
String  dataSent = new String("message to be sent");
mySendMessage.writeBytes(dataSent.getBytes());
```

Next, the message is sent using the sender service `mySender`.

Sending a message

```
mySender.send(mySendMessage);
```

The policy used is either the default policy for the service, if specified, or the system default policy. The message attributes are set from the policy or service, or the default for the messaging transport.

When more control is needed you can pass a policy object:

Sending a message with a specified policy

```
mySender.send(mySendMessage, myPolicy);
```

The policy controls the behavior of the send command. In particular, the policy specifies whether the send is part of a unit of work, the priority, persistence and expiry of the message and whether policy components should be invoked. Whether the queue should be implicitly opened and left open can also be controlled.

To send a message to a distribution list, for instance `myDistList`, use it as the sender service:

Sending a message to a distribution list

```
myDistList.send(mySendMessage);
```

You can set an attribute such as the *Format* before the message is sent, to override the default in the policy or service.

Setting an attribute in a message

```
mySendMessage.setFormat(myFormat):
```

Similarly, after a message has been sent you can retrieve an attribute such as the *MessageID*.

Getting an attribute from a message

```
msgId = mySendMessage.getMessageId();
```

For details of the message attributes that you can set and get, see “AmMessage” on page 210.

When a message object is used to send a message, it might not be left in the same state as it was prior to the send. Therefore, if you use the message object for repeated send operations, it is advisable to reset it to its initial state (see **reset** on page 233) and rebuild it each time.

Sample program

For more details, refer to the `SendAndForget.java` sample program (see “The sample programs” on page 277).

Receiving messages

The next example shows how to receive a message from the receiver service `myReceiver`, and to read the data from the message object `myReceiveMessage`.

Receiving a message and retrieving the data

```
myReceiver.receive(myReceiveMessage);
data = myReceiveMessage.readBytes(myReceiveMessage.getDataLength());
```

The policy used will be the default for the service if defined, or the system default policy. Greater control of the behavior of the receive can be achieved by passing a policy object.

Receiving a message with a specified policy

```
myReceiver.receive(myReceiveMessage, myPolicy);
```

The policy can specify the wait interval, whether the call is part of a unit of work, whether the message should be code page converted, whether all the members of a group must be there before any members can be read, and how to deal with backout failures.

To receive a specific message using its correlation ID, create a selection message object and set its *CorrelId* attribute to the required value. The selection message is then passed as a parameter on the receive.

Receiving a specific message using the correlation ID

```
mySelectionMessage = mySession.createMessage("MY.SELECTION.MESSAGE");
mySelectionMessage.setCorrelationId(myCorrelId);
myReceiver.receive(myReceiveMessage, mySelectionMessage, myPolicy);
```

As before, the policy is optional.

You can view the attributes of the message just received, such as the *Encoding*.

Getting an attribute from the message

```
encoding = myReceiveMessage.getEncoding();
```

Sample program

For more details, refer to the `Receiver.java` sample program (see "The sample programs" on page 277).

Request/response messaging

In the *request/response* style of messaging, a requester (or client) application sends a request message and expects to receive a response message back. The responder (or server) application receives the request message and produces the response message (or messages) which it sends back to the requester application. The responder application uses information in the request message to know how to send the response message back to the requester.

In the following examples 'your' refers to the responding application (the server); 'my' refers to the requesting application (the client).

The requester sends a message as described in "Sending messages" on page 196, specifying the service (`myReceiver`) to which the response message should be sent.

Sending a request message

```
mySender.send(mySendMessage, myReceiver);
```

A policy object can also be specified if required.

The responder receives the message as described in "Receiving messages" on page 197, using its receiver service (`yourReceiver`). It also receives details of the response service (`yourResponder`) for sending the response.

Receiving the request message

```
yourReceiver.receive(yourReceiveMessage, yourResponder);
```

A policy object can be specified if required, as can a selection message object (see "Receiving messages" on page 197).

The responder sends its response message (`yourReplyMessage`) to the response service, specifying the received message to which this is a response.

Sending a response to the request message

```
yourResponder.send(yourReplyMessage, yourReceiveMessage);
```

Finally, the requester application receives the response (`myResponseMessage`), which is correlated with the original message it sent (`mySendMessage`).

Receiving the response message

```
myReceiver.receive(myResponseMessage, mySendMessage);
```

In a typical application the responder might be a server operating in a loop, receiving requests and replying to them. In this case, the message objects should be set to their initial state and the data cleared before servicing the next request. This is achieved as follows:

Resetting the message object

```
yourReceiveMessage.reset();
yourResponseMessage.reset();
```

Sample programs

For more details, refer to the `Client.java` and `Server.java` sample programs (see "The sample programs" on page 277).

Publish/subscribe messaging

With *publish/subscribe* messaging a *publisher* application publishes messages to *subscriber* applications using a *broker*. The message published contains application data and one or more *topic* strings that describe the data. A subscribing application subscribes to topics informing the broker which topics it is interested in. When the broker receives a message from a publisher it compares the topics in the messages to the topics in the subscription from subscribing applications. If they match, the broker forwards the message to the subscribing application.

Data on a particular topic is published as shown in the next example.

Publishing a message on a specified topic

```
String publicationTopic = new String("Weather");
String publicationData = new String("The weather is sunny");

myPubMessage.addTopic(publicationTopic);
myPubMessage.writeBytes(publicationData.getBytes());
myPublisher.publish(myPubMessage, myReceiver);
```

`myReceiver` identifies a response service to which the broker will send any response messages. You can also specify a policy object to modify the behavior of the command.

To subscribe to a publish/subscribe broker you need to specify one or more topics.

Subscribing to a broker on specified topics

```
String weather = new String("Weather");
String birds = new String("Birds");

mySubMessage.addTopic(weather);
mySubMessage.addTopic(birds);
mySubscriber.subscribe(mySubMessage, myReceiver);
```

Broker response messages will be sent to `myReceiver`.

To remove a subscription, add the topic or topics to be deleted to the message object, and use:

Removing a subscription

```
mySubscriber.unsubscribe(myUnsubMessage, myReceiver);
```

To receive a publication from a broker, use:

Receiving a publication

```
mySubscriber.receive(myReceiveMessage, myPolicy);
publication = myReceiveMessage.readBytes(
                 myReceiveMessage.getDataLength());
```

You can then use the **getTopicCount** and **getTopic** methods to extract the topic or topics from the message object.

Sample programs

For more details, refer to the `Publisher.java` and `Subscriber.java` sample programs (see “The sample programs” on page 277).

Using AmElement objects

Publish/subscribe brokers (such as MQSeries Publish/Subscribe) respond to messages that contain name/value pairs to define the commands and options to be carried out. The Application Messaging Interface contains some methods which produce these name/value pairs directly (such as **AmSubscriber.subscribe**). For less commonly used commands, the name/value pairs can be added to a message using an AmElement object.

For example, to send a message containing a ‘Request Update’ command, use the following:

Using an AmElement object to construct a command message

```
AmElement bespokeElement = new AmElement("MQPSCommand", "ReqUpdate");
mySendMessage.addElement(bespokeElement);
```

You must then send the message, using **AmSender.send**, to the sender service specified for your publish/subscribe broker.

If you use streams with MQSeries Publish/Subscribe, you must add the appropriate name/value element explicitly to the message object.

The message element methods can, in fact, be used to add any element to a message before issuing an publish/subscribe request. Such elements (including topics, which are specialized elements) supplement or override those added implicitly by the request, as appropriate to the individual element type.

The use of name/value elements is not restricted to publish/subscribe applications, they can be used in other applications as well.

Error handling

The **getLastErrorStatus** method always reflects the last most severe error experienced by an object. It can be used to return an AmStatus object encapsulating this error state. Once the error state has been handled, **clearErrorCodes** can be called to reset this error state.

AmJava can raise two types of Exception, one to reflect serious errors and the other to reflect warnings. By default, only AmErrorExceptions are raised. AmWarningExceptions can be enabled using the **enableWarnings** method. Since both are types of AmException, a generic catch block can be used to process all amJava Exceptions.

Enabling AmWarningExceptions might have some unexpected side-effects, especially when an AmObject is returning data such as another AmObject. For example, if AmWarningExceptions are enabled for an AmSession object and an AmSender is created that does not exist in the repository, an AmWarningException will be raised to reflect this fact. If this happens, the AmSender object will not be created since its creation was interrupted by an Exception. However, there might be times during the life of an AmObject when processing AmWarningExceptions is useful.

For example:

```
try
{
    ...
    mySession.enableWarnings(true);
    mySession.open();
    ...
}
catch (AmErrorException errorEx)
{
    AmStatus sessionStatus = mySession.getLastErrorStatus();
    switch (sessionStatus.getReasonCode())
    {
    case AmConstants.AMRC_XXXX:
        ...
    case AmConstants.AMRC_XXXX:
        ...
    }
    mySession.clearErrorCodes();
}
catch (AmWarningException warningEx)
{
    ...
}
```

Since most of the objects implement the AmObject interface, a generic error handling routine can be written. For example:

```
try
{
    ...
    mySession.open();
    ...
    mySender.send(myMessage):
    ...
    mySender.send(myMessage):
    ...
    mySession.commit();
}
catch(AmException amex);
{
    AmStatus status;
    status = amex.getSource().getLastErrorStatus();
    System.out.println("Object in error; name="+ amex.getSource().getName());
    System.out.println("Object in error; RC="+ status.getReasonCode());
    ...
    amex.getSource().clearErrorCodes();
}
```

The catch block works because all objects that throw the AmException in the try block are AmObjects, and so they all have **getName**, **getLastErrorStatus** and **clearErrorCodes** methods.

Transaction support

Messages sent and received by the AMI can, optionally, be part of a transactional unit of work. A message is included in a unit of work based on the setting of the syncpoint attribute specified in the policy used on the call. The scope of the unit of work is the session handle and only one unit of work may be active at any time.

The API calls used to control the transaction depends on the type of transaction is being used.

- MQSeries messages are the only resource

  A transaction is started by the first message sent or received under syncpoint control, as specified in the policy specified for the send or receive. Multiple messages can be included in the same unit of work. The transaction is committed or backed out using the **commit** or **rollback** method.

- Using MQSeries as an XA transaction coordinator

  The transaction must be started explicitly using the **begin** method before the first recoverable resource (such as a relational database) is changed. The transaction is committed or backed out using an **commit** or **rollback** method.

- Using an external transaction coordinator

  The transaction is controlled using the API calls of an external transaction coordinator (such as CICS, Encina or Tuxedo). The AMI calls are not used but the syncpoint attributed must still be specified in the policy used on the call.

Other considerations

Multithreading

If you are using multithreading with the AMI, a session normally remains locked for the duration of a single AMI call. If you use receive with wait, the session remains locked for the duration of the wait, which might be unlimited (that is, until the wait time is exceeded or a message arrives on the queue). If you want another thread to run while a thread is waiting for a message, it must use a separate session.

AMI handles and object references can be used on a different thread from that on which they were first created for operations that do not involve an access to the underlying (MQSeries) message transport. Functions such as initialize, terminate, open, close, send, receive, publish, subscribe, unsubscribe, and receive publication will access the underlying transport restricting these to the thread on which the session was first opened (for example, using **AmSession.open**). An attempt to issue these on a different thread will cause an error to be returned by MQSeries and a transport error (AMRC_TRANSPORT_ERR) will be reported to the application.

Using MQSeries with the AMI

You must not mix MQSeries function calls with AMI calls within the same process.

Field limits

When string and binary properties such as queue name, message format, and correlation ID are set, the maximum length values are determined by MQSeries, the underlying message transport. See the rules for naming MQSeries objects in the *MQSeries Application Programming Guide*.

Building Java applications

AMI package for Java

AMI provides a jar file that contains all the classes comprising the AMI package for Java.

com.ibm.mq.amt	Java package
com.ibm.mq.amt.jar	Java jar file

This jar file is installed under:

```
/java/lib        (UNIX)

\java\lib        (Windows)
```

See "Directory structure" on page 263 (AIX), 267 (Solaris), or 270 (Windows).

In order to make use of this package you must:

- Import the package into your Java application by using the following statement in that application:

  ```
  import com.ibm.mq.amt.*;
  ```

- Make sure the AMI jar file is in your CLASSPATH environment variable. See "Setting the runtime environment" on page 262 (AIX), page 266 (Solaris), or page 269 (Windows).

  This should be done both in the environment in which your Java program is compiled, and the environment in which it is run.

Running Java programs

This section explains what you have to do to prepare and run your Java programs on the AIX, Sun Solaris, Windows 98 and Windows NT operating systems.

The AMI interface for Java makes use of JNI (Java Native Interface) and so requires a platform native library to run successfully. This library must be accessible to your runtime environment. See "Language compilers" on page 260 for versions of the Java Developer's Kit (JDK) supported by the AMI.

AIX

Make sure that the JNI library `libamtJava.so` is accessible to your runtime environment. If the **amtInstall** utility has been run, this will have been done for you (see "Installation on AIX" on page 261). If you have not run the utility, you should perform:

```
ln -s /usr/mqm/lib/libamtJava.so /usr/lib/libamtJava.so
```

Solaris

Make sure that the JNI library `libamtJava.so` is accessible to your runtime environment. If the **amtInstall** utility has been run, this will have been done for you (see "Installation on Sun Solaris" on page 265). If you have not run the utility, you should perform:

```
ln -s /opt/mqm/lib/libamtJava.so /usr/lib/libamtJava.so
```

Windows

Make sure that the JNI library `amtJava.dll` is in one of the directories specified in the PATH environment variable for your runtime environment. For example:

```
SET PATH=%PATH%;C:\MQSeries\bin;
```

If you already have MQSeries installed, it is likely that this environment has already been set up for you.

Once the AMI jar file and the JNI library are referenced in your runtime environment you can run your Java application. For example, to run an application called `mine` that exists in a package `com.xxx.com`, perform:

```
java com.xxx.com.mine
```

Chapter 10. Java interface overview

This chapter contains an overview of the structure of the Application Messaging Interface for Java. Use it to find out what functions are available in this interface.

The Java interface provides sets of methods for each of the classes listed below. The methods available for each class are listed in the following pages. Follow the page references to see the reference information for each method.

Base classes

AmSessionFactory	page 208
AmSession	page 209
AmMessage	page 210
AmSender	page 212
AmReceiver	page 213
AmDistributionList	page 214
AmPublisher	page 215
AmSubscriber	page 216
AmPolicy	page 217

Helper classes

AmConstants	page 218
AmElement	page 218
AmObject	page 218
AmStatus	page 218

Exception classes

AmException	page 219
AmErrorException	page 219
AmWarningExcpetion	page 219

AmSessionFactory

The **AmSessionFactory** class is used to create AmSession objects.

Constructor

Constructor for AmSessionFactory.

AmSessionFactory	page 222

Session factory management

Methods to return the name of an AmSessionFactory object, and to control traces.

getFactoryName	page 222
getLocalHost	page 222
getRepository	page 222
getTraceLevel	page 222
getTraceLocation	page 222
setLocalHost	page 223
setRepository	page 223
setTraceLevel	page 223
setTraceLocation	page 223

Create session

Method to create an AmSession object.

createSession	page 222

AmSession

The **AmSession** object creates and manages all other objects, and provides scope for a unit of work.

Session management

Methods to open and close an AmSession object, to return its name, and to control traces.

open	page 227
close	page 224
getName	page 226
getTraceLevel	page 226
getTraceLocation	page 227

Create objects

Methods to create AmMessage, AmSender, AmReceiver, AmDistributionList AmPublisher, AmSubscriber, and AmPolicy objects.

createMessage	page 225
createSender	page 226
createReceiver	page 225
createDistributionList	page 225
createPublisher	page 225
createSubscriber	page 226
createPolicy	page 225

Transactional processing

Methods to begin, commit and rollback a unit of work.

begin	page 224
commit	page 224
rollback	page 227

Error handling

Methods to clear the error codes, enable warnings, and return the status from the last error.

clearErrorCodes	page 224
enableWarnings	page 226
getLastErrorStatus	page 226

AmMessage

An **AmMessage** object encapsulates an MQSeries message descriptor (MQMD) structure, and it contains the message data if this is not passed as a separate parameter.

Get values

Methods to get the CCSID, correlation ID, encoding, format, group status, message ID and name of the message object.

getCCSID	page 229
getCorrelationId	page 230
getEncoding	page 230
getFormat	page 231
getGroupStatus	page 231
getMessageId	page 231
getName	page 231

Set values

Methods to set the CCSID, correlation ID, format and group status of the message object.

setCCSID	page 233
setCorrelationId	page 233
setEncoding	page 233
setFormat	page 234
setGroupStatus	page 234

Reset values

Method to reset the message object to the state it had when first created.

reset	page 233

Read and write data

Methods to read or write byte data to or from the message object, to get and set the data offset, and to get the length of the data.

getDataLength	page 230
getDataOffset	page 230
setDataOffset	page 233
readBytes	page 232
writeBytes	page 234

Publish/subscribe topics

Methods to manipulate the topics in a publish/subscribe message.

addTopic	page 228
deleteTopic	page 229
getTopic	page 232
getTopicCount	page 232

Publish/subscribe name/value elements

Methods to manipulate the name/value elements in a publish/subscribe message.

addElement	page 228
deleteElement	page 229
getElement	page 230
getElementCount	page 230
deleteNamedElement	page 229
getNamedElement	page 232
getNamedElementCount	page 232

Error handling

Methods to clear the error codes, enable warnings, and return the status from the last error.

clearErrorCodes	page 228
enableWarnings	page 229
getLastErrorStatus	page 231

AmSender

An **AmSender** object encapsulates an MQSeries object descriptor (MQOD) structure.

Open and close

Methods to open and close the sender service.

open	page 236
close	page 235

Send

Method to send a message.

send	page 236

Get values

Methods to get the CCSID, encoding and name of the sender service.

getCCSID	page 235
getEncoding	page 236
getName	page 236

Error handling

Methods to clear the error codes, enable warnings, and return the status from the last error.

clearErrorCodes	page 235
enableWarnings	page 235
getLastErrorStatus	page 236

AmReceiver

An **AmReceiver** object encapsulates an MQSeries object descriptor (MQOD) structure.

Open and close

Methods to open and close the receiver service.

open	page 239
close	page 238

Receive and browse

Methods to receive or browse a message.

receive	page 239
browse	page 237

Get values

Methods to get the definition type, name and queue name of the receiver service.

getDefinitionType	page 238
getName	page 239
getQueueName	page 239

Set value

Method to set the queue name of the receiver service.

setQueueName	page 239

Error handling

Methods to clear the error codes, enable warnings, and return the status from the last error.

clearErrorCodes	page 238
enableWarnings	page 238
getLastErrorStatus	page 238

AmDistributionList

An **AmDistributionList** object encapsulates a list of AmSender objects.

Open and close

Methods to open and close the distribution list service.

open	page 241
close	page 240

Send

Method to send a message to the distribution list.

send	page 241

Get values

Methods to get the name of the distribution list service, a count of the AmSenders in the list, and one of the AmSenders that is contained in the list.

getName	page 240
getSenderCount	page 241
getSender	page 240

Error handling

Methods to clear the error codes, enable warnings, and return the status from the last error.

clearErrorCodes	page 240
enableWarnings	page 240
getLastErrorStatus	page 240

AmPublisher

An **AmPublisher** object encapsulates a sender service and provides support for publishing messages to a publish/subscribe broker.

Open and close

Methods to open and close the publisher service.

open	page 243
close	page 242

Publish

Method to publish a message.

publish	page 243

Get values

Methods to get the CCSID, encoding and name of the publisher service.

getCCSID	page 242
getEncoding	page 242
getName	page 243

Error handling

Methods to clear the error codes, enable warnings, and return the status from the last error.

clearErrorCodes	page 242
enableWarnings	page 242
getLastErrorStatus	page 242

AmSubscriber

An **AmSubscriber** object encapsulates both a sender service and a receiver service. It provides support for subscribe and unsubscribe requests to a publish/subscribe broker, and for receiving publications from the broker.

Open and close

Methods to open and close the subscriber service.

open	page 245
close	page 244

Broker messages

Methods to subscribe to a broker, remove a subscription, and receive a publication from the broker.

subscribe	page 247
unsubscribe	page 247
receive	page 246

Get values

Methods to get the CCSID, definition type, encoding, name and queue name of the subscriber service.

getCCSID	page 244
getDefinitionType	page 244
getEncoding	page 245
getName	page 245
getQueueName	page 245

Set value

Method to set the queue name of the subscriber service.

setQueueName	page 246

Error handling

Methods to clear the error codes, enable warnings, and return the status from the last error.

clearErrorCodes	page 244
enableWarnings	page 244
getLastErrorStatus	page 245

AmPolicy

An **AmPolicy** object encapsulates the options used during AMI operations.

Policy management

Methods to return the name of the policy, and to get and set the wait time when receiving a message.

getName	page 248
getWaitTime	page 248
setWaitTime	page 248

Error handling

Methods to clear the error codes, enable warnings, and return the status from the last error.

clearErrorCodes	page 248
enableWarnings	page 248
getLastErrorStatus	page 248

Helper classes

A Java Interface, and classes that encapsulate constants, name/value elements, and error status.

AmConstants

Provides access to all the AMI constants.

AmConstants	page 249

AmElement

Constructor for AmElement, and methods to return the name, type, value and version of an element, to set the version, and to return a String representation of the element.

AmElement	page 250
getName	page 250
getValue	page 250
getVersion	page 250
setVersion	page 250
toString	page 250

AmObject

A Java Interface containing methods to return the name of the object, to clear the error codes and to return the last error condition.

clearErrorCodes	page 251
getLastErrorStatus	page 251
getName	page 251

AmStatus

Constructor for AmStatus, and methods to return the completion code, reason code, secondary reason code and status text, and to return a String representation of the AmStatus.

AmStatus	page 252
getCompletionCode	page 252
getReasonCode	page 252
getReasonCode2	page 252
toString	page 252

Exception classes

Classes that encapsulate error and warning conditions. AmErrorException and AmWarningException inherit from AmException.

AmException

Methods to return the completion code and reason code from the Exception, the class name, method name and source of the Exception, and to return a String representation of the Exception.

getClassName	page 253
getCompletionCode	page 253
getMethodName	page 253
getReasonCode	page 253
getSource	page 253
toString	page 253

AmErrorException

Methods to return the completion code and reason code from the Exception, the class name, method name and source of the Exception, and to return a String representation of the Exception.

getClassName	page 254
getCompletionCode	page 254
getMethodName	page 254
getReasonCode	page 254
getSource	page 254
toString	page 254

AmWarningException

Methods to return the completion code and reason code from the Exception, the class name, method name and source of the Exception, and to return a String representation of the Exception.

getClassName	page 255
getCompletionCode	page 255
getMethodName	page 255
getReasonCode	page 255
getSource	page 255
toString	page 255

Chapter 11. Java interface reference

In the following sections the Java interface methods are listed by the class they refer to. Within each section the methods are listed in alphabetical order.

Note that where constants are shown (for example, AMRC_NONE), they can be accessed using the AmConstants class (for example, AmConstants.AMRC_NONE). See page 249.

Base classes

Note that all of the methods in these classes can throw AmWarningException and AmErrorException (see below). However, by default, AmWarningExceptions are not raised.

AmSessionFactory

The **AmSessionFactory** class is used to create AmSession objects.

AmSessionFactory

Constructor for an AmSessionFactory.

```
AmSessionFactory(String name);
```

`name`	The name of the AmSessionFactory. This is the location of the data files used by the AMI (the repository file and the local host file). The name can be a fully qualified directory that includes the path under which the files are located. Otherwise, see “Local host and repository files” on page 272 for the location of these files.

createSession

Creates an AmSession object.

```
AmSession  createSession(String name);
```

`name`	The name of the AmSession.

getFactoryName

Returns the name of the AmSessionFactory.

```
String  getFactoryName();
```

getLocalHost

Returns the name of the local host file.

```
String  getLocalHost();
```

getRepository

Returns the name of the repository file.

```
String  getRepository();
```

getTraceLevel

Returns the trace level for the AmSessionFactory.

```
int  getTraceLevel();
```

getTraceLocation

Returns the location of the trace for the AmSessionFactory.

```
String  getTraceLocation();
```

setLocalHost

Sets the name of the AMI local host file to be used by any AmSession created from this AmSessionFactory. (Otherwise, the default host file `amthost.xml` is used.)

```
void  setLocalHost(String  fileName);
```

`fileName`	The name of the file used by the AMI as the local host file. This file must be present on the local file system or an error will be produced upon the creation of an AmSession.

setRepository

Sets the name of the AMI repository to be used by any AmSession created from this AmSessionFactory. (Otherwise, the default repository file `amt.xml` is used.)

```
void  setRepository(String  fileName);
```

`fileName`	The name of the file used by the AMI as the repository. This file must be present on the local file system or an error will be produced upon the creation of an AmSession.

setTraceLevel

Sets the trace level for the AmSessionFactory.

```
void  setTraceLevel(int level);
```

`level`	The trace level to be set in the AmSessionFactory. Trace levels are 0 through 9, where 0 represents minimal tracing and 9 represents a fully detailed trace.

setTraceLocation

Sets the location of the trace for the AmSessionFactory.

```
void  setTraceLocation(String location);
```

`location`	The location on the local system where trace files will be written. This location must be a directory, and it must exist prior to the trace being run.

AmSession

An **AmSession** object provides the scope for a unit of work and creates and manages all other objects, including at least one connection object. Each (MQSeries) connection object encapsulates a single MQSeries queue manager connection. The session object definition specifying the required queue manager connection can be provided by a repository policy definition, or by default will name a single local queue manager with no repository. The session, when deleted, is responsible for releasing memory by closing and deleting all other objects that it manages.

begin

Begins a unit of work in this AmSession, allowing an AMI application to take advantage of the resource coordination provided in MQSeries Version 5. The unit of work can subsequently be committed by the **commit** method, or backed out by the **rollback** method. This should be used only when AMI is the transaction coordinator. If available, native coordination APIs (for example CICS or Tuxedo) should be used.

begin is overloaded. The `policy` parameter is optional.

```
void  begin(AmPolicy policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

clearErrorCodes

Clears the error codes in the AmSession.

```
void  clearErrorCodes();
```

close

Closes the AmSession, and all open objects owned by it. **close** is overloaded: the `policy` parameter is optional.

```
void  close(AmPolicy policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

commit

Commits a unit of work that was started by **AmSession.begin**. **commit** is overloaded: the `policy` parameter is optional.

```
void  commit(AmPolicy policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

createDistributionList

Creates an AmDistributionList object.

```
AmDistributionList  createDistributionList(String name);
```

`name`	The name of the AmDistributionList. This must match the name of a distribution list defined in the repository.

createMessage

Creates an AmMessage object.

```
AmMessage  createMessage(String name);
```

`name`	The name of the AmMessage. This can be any name that is meaningful to the application.

createPolicy

Creates an AmPolicy object.

```
AmPolicy  createPolicy(String name);
```

`name`	The name of the AmPolicy. If it matches a policy defined in the repository, the policy will be created using the repository definition, otherwise it will be created with default values.

createPublisher

Creates an AmPublisher object.

```
AmPublisher  createPublisher(String name);
```

`name`	The name of the AmPublisher. If it matches a publisher defined in the repository, the publisher will be created using the repository definition, otherwise it will be created with default values (that is, with an AmSender name that matches the publisher name).

createReceiver

Creates an AmReceiver object.

```
AmReceiver  createReceiver(String name);
```

`name`	The name of the AmReceiver. If it matches a receiver defined in the repository, the receiver will be created using the repository definition, otherwise it will be created with default values (that is, with a queue name that matches the receiver name).

createSender

Creates an AmSender object.

```
AmSender  createSender(String name);
```

`name`	The name of the AmSender. If it matches a sender defined in the repository, the sender will be created using the repository definition, otherwise it will be created with default values (that is, with a queue name that matches the sender name).

createSubscriber

Creates an AmSubscriber object.

```
AmSubscriber  createSubscriber(String name);
```

`name`	The name of the AmSubscriber. If it matches a subscriber defined in the repository, the subscriber will be created using the repository definition, otherwise it will be created with default values (that is, with an AmSender name that matches the subscriber name, and an AmReceiver name that is the same with the addition of the suffix '.RECEIVER').

enableWarnings

Enables AmWarningExceptions; the default value for any AmObject is that AmWarningExceptions are not raised. Note that warning reason codes can be retrieved using **getLastErrorStatus**, even if AmWarningExceptions are disabled.

```
void  enableWarnings(boolean warningsOn);
```

`warningsOn`	If set to `true`, AmWarningExceptions will be raised for this object.

getLastErrorStatus

Returns the AmStatus of the last error condition.

```
AmStatus  getLastErrorStatus();
```

getName

Returns the name of the AmSession.

```
String  getName();
```

getTraceLevel

Returns the trace level of the AmSession.

```
int  getTraceLevel();
```

getTraceLocation

Returns the location of the trace for the AmSession.

```
  String  getTraceLocation();
```

open

Opens an AmSession using the specified policy. The application profile group of this policy provides the connection definitions enabling the connection objects to be created. The specified library is loaded for each connection and its dispatch table initialized. If the transport type is MQSeries and the MQSeries local queue manager library cannot be loaded, then the MQSeries client queue manager is loaded. Each connection object is then opened.

open is overloaded: the `policy` parameter is optional.

```
  void  open(AmPolicy policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

rollback

Rolls back a unit of work that was started by **AmSession.begin**, or under policy control. **rollback** is overloaded: the `policy` parameter is optional.

```
  void  rollback(AmPolicy policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

AmMessage

An **AmMessage** object encapsulates the MQSeries MQMD message properties, and name/value elements such as the topics for publish/subscribe messages. In addition it contains the application data.

The initial state of the message object is:

`CCSID`	default queue manager CCSID
`correlationId`	all zeroes
`dataLength`	zero
`dataOffset`	zero
`elementCount`	zero
`encoding`	AMENC_NATIVE
`format`	AMFMT_STRING
`groupStatus`	AMGRP_MSG_NOT_IN_GROUP
`topicCount`	zero

When a message object is used to send a message, it might not be left in the same state as it was prior to the send. Therefore, if you use the message object for repeated send operations, it is advisable to reset it to its initial state (see **reset** on page 233) and rebuild it each time.

addElement

Adds a name/value element to an AmMessage object. **addElement** is overloaded: the `element` parameter is required, but the `options` parameter is optional.

```
  void  addElement(
    AmElement  element,
    int        options);
```

`element`	The element to be added to the AmMessage.
`options`	The options to be used. This parameter is reserved and must be set to zero.

addTopic

Adds a publish/subscribe topic to an AmMessage object.

```
  void  addTopic(String topicName);
```

`topicName`	The name of the topic to be added to the AmMessage.

clearErrorCodes

Clears the error in the AmMessage object.

```
  void  clearErrorCodes();
```

deleteElement

Deletes the element in the AmMessage object at the specified index. Indexing is within all elements of a message, and might include topics (which are specialized elements).

```
  void  deleteElement(int index);
```

`index`	The index of the element to be deleted, starting from zero. On completion, elements with higher `index` values than that specified will have those values reduced by one. **getElementCount** gets the number of elements in the message.

deleteNamedElement

Deletes the element with the specified name in the AmMessage object, at the specified index. Indexing is within all elements that share the same name.

```
  void  deleteNamedElement(
    String  name,
    int     index);
```

`name`	The name of the element to be deleted.
`index`	The index of the element to be deleted, starting from zero. On completion, elements with higher `index` values than that specified will have those values reduced by one. **getNamedElementCount** gets the number of elements in the message with the specified name.

deleteTopic

Deletes a publish/subscribe topic in an AmMessage object at the specified index. Indexing is within all topics in the message.

```
  void  deleteTopic(int index);
```

`index`	The index of the topic to be deleted, starting from zero. **getTopicCount** gets the number of topics in the message.

enableWarnings

Enables AmWarningExceptions; the default value for any AmObject is that AmWarningExceptions are not raised. Note that warning reason codes can be retrieved using **getLastErrorStatus**, even if AmWarningExceptions are disabled.

```
  void  enableWarnings(boolean warningsOn);
```

`warningsOn`	If set to `true`, AmWarningExceptions will be raised for this object.

getCCSID

Returns the coded character set identifier used by AmMessage.

```
  int  getCCSID();
```

getCorrelationId

Returns the correlation identifier for the AmMessage.

```
byte[]  getCorrelationId();
```

getDataLength

Returns the length of the message data in the AmMessage.

```
int  getDataLength();
```

getDataOffset

Returns the current offset in the message data for reading or writing data bytes.

```
int  getDataOffset();
```

getElement

Returns an element in an AmMessage object at the specified index. Indexing is within all elements in the message, and might include topics (which are specialized elements).

```
AmElement  getElement(int index);
```

`index`	The index of the element to be returned, starting from zero. **getElementCount** gets the number of elements in the message.

getElementCount

Returns the total number of elements in an AmMessage object. This might include topics (which are specialized elements).

```
int  getElementCount();
```

getEncoding

Returns the value used to encode numeric data types for the AmMessage.

```
int  getEncoding();
```

The following values can be returned:

```
AMENC_NORMAL
AMENC_NORMAL_FLOAT_390
AMENC_REVERSED
AMENC_REVERSED_FLOAT_390
AMENC_UNDEFINED
```

getFormat

Returns the format of the AmMessage.

```
String  getFormat();
```

The following values can be returned:

```
AMFMT_NONE
AMFMT_STRING
AMFMT_RF_HEADER
```

getGroupStatus

Returns the group status value for the AmMessage. This indicates whether the message is in a group, and if it is the first, middle, last or only one in the group.

```
int  getGroupStatus();
```

The following values can be returned:

```
AMGRP_MSG_NOT_IN_GROUP
AMGRP_FIRST_MSG_IN_GROUP
AMGRP_MIDDLE_MSG_IN_GROUP
AMGRP_LAST_MSG_IN_GROUP
AMGRP_ONLY_MSG_IN_GROUP
```

Alternatively, bitwise tests can be performed using the constants:

```
AMGF_IN_GROUP
AMGF_FIRST
AMGF_LAST
```

getLastErrorStatus

Returns the AmStatus of the last error condition for this object.

```
AmStatus  getLastErrorStatus();
```

getMessageId

Returns the message identifier from the AmMessage object.

```
byte[]  getMessageId();
```

getName

Returns the name of the AmMessage object.

```
String  getName();
```

getNamedElement

Returns the element with the specified name in an AmMessage object, at the specified index. Indexing is within all elements that share the same name.

```
AmElement  getNamedElement(
  String  name,
  int     index);
```

`name`	The name of the element to be returned.
`index`	The index of the element to be returned, starting from zero.

getNamedElementCount

Returns the total number of elements with the specified name in the AmMessage object.

```
int  getNamedElementCount(String name);
```

`name`	The name of the elements to be counted.

getTopic

Returns the publish/subscribe topic in the AmMessage object, at the specified index. Indexing is within all topics.

```
String  getTopic(int index);
```

`index`	The index of the topic to be returned, starting from zero. **getTopicCount** gets the number of topics in the message.

getTopicCount

Returns the total number of publish/subscribe topics in the AmMessage object.

```
int  getTopicCount();
```

readBytes

Populates a byte array with data from the AmMessage, starting at the current data offset (which must be positioned before the end of the data for the read to be successful). Use **setDataOffset** to specify the data offset. **readBytes** will advance the data offset by the number of bytes read, leaving the offset immediately after the last byte read.

```
byte[]  readBytes(int dataLength);
```

`dataLength`	The maximum number of bytes to be read from the message data. The number of bytes returned is the minimum of `dataLength` and the number of bytes between the data offset and the end of the data.

reset

Resets the AmMessage object to its initial state (see page 228).

reset is overloaded: the `options` parameter is optional.

```
void  reset(int options);
```

`options`	A reserved field that must be set to zero.

setCCSID

Sets the coded character set identifier used by the AmMessage object.

```
void  setCCSID(int codedCharSetId);
```

`codedCharSetId`	The CCSID to be set in the AmMessage.

setCorrelationId

Sets the correlation identifier in the AmMessage object.

```
void  setCorrelationId(byte[] correlId);
```

`correlId`	The correlation identifier to be set in the AmMessage.

setDataOffset

Sets the data offset for reading or writing byte data.

```
void  setDataOffset(int dataOffset);
```

`dataOffset`	The data offset to be set in the AmMessage. Set an offset of zero to read or write from the start of the data.

setEncoding

Sets the encoding of the data in the AmMessage object.

```
void  setEncoding(int encoding);
```

`encoding` The encoding to be used in the AmMessage. It can take one of the following values:

```
AMENC_NORMAL
AMENC_NORMAL_FLOAT_390
AMENC_REVERSED
AMENC_REVERSED_FLOAT_390
AMENC_UNDEFINED
```

setFormat

Sets the format for the AmMessage object.

```
void  setFormat(String format);
```

`format`	The format to be used in the AmMessage. It can take one of the following values:
	`AMFMT_NONE`
	`AMFMT_STRING`
	`AMFMT_RF_HEADER`
	If set to AMFMT_NONE, the default format for the sender will be used (if available).

setGroupStatus

Sets the group status value for the AmMessage. This indicates whether the message is in a group, and if it is the first, middle, last or only one in the group. Once you start sending messages in a group, you must complete the group before sending any messages that are not in the group.

If you specify AMGRP_MIDDLE_MSG_IN_GROUP or AMGRP_LAST_MSG_IN_GROUP without specifying AMGRP_FIRST_MSG_IN_GROUP, the behaviour is the same as for AMGRP_FIRST_MSG_IN_GROUP and AMGRP_ONLY_MSG_IN_GROUP.

If you specify AMGRP_FIRST_MSG_IN_GROUP out of sequence, then the behaviour is the same as for AMGRP_MIDDLE_MSG_IN_GROUP.

```
void  setGroupStatus(int groupStatus);
```

`groupStatus`	The group status to be set in the AmMessage. It can take one of the following values:
	`AMGRP_MSG_NOT_IN_GROUP`
	`AMGRP_FIRST_MSG_IN_GROUP`
	`AMGRP_MIDDLE_MSG_IN_GROUP`
	`AMGRP_LAST_MSG_IN_GROUP`
	`AMGRP_ONLY_MSG_IN_GROUP`

writeBytes

Writes a byte array into the AmMessage object, starting at the current data offset. If the data offset is not at the end of the data, existing data is overwritten. Use **setDataOffset** to specify the data offset. **writeBytes** will advance the data offset by the number of bytes written, leaving it immediately after the last byte written.

```
void  writeBytes(byte[] data);
```

`data`	The data to be written to the AmMessage.

AmSender

An **AmSender** object encapsulates an MQSeries object descriptor (MQOD) structure. This represents an MQSeries queue on a local or remote queue manager. An open sender service is always associated with an open connection object (such as a queue manager connection). Support is also included for dynamic sender services (those that encapsulate model queues). The required sender service object definitions can be provided from a repository, or created without a repository definition by defaulting to the existing queue objects on the local queue manager.

The AmSender object must be created before it can be opened. This is done using **AmSession.createSender**.

A *responder* is a special type of AmSender used for sending a response to a request message. It is not created from a repository definition. Once created, it must not be opened until used in its correct context as a responder receiving a request message with **AmReceiver.receive**. When opened, its queue and queue manager properties are modified to reflect the *ReplyTo* destination specified in the message being received. When first used in this context, the sender service becomes a responder sender service.

clearErrorCodes

Clears the error codes in the AmSender.

```
void  clearErrorCodes();
```

close

Closes the AmSender. **close** is overloaded: the `policy` parameter is optional.

```
void  close(AmPolicy policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

enableWarnings

Enables AmWarningExceptions; the default value for any AmObject is that AmWarningExceptions are not raised. Note that warning reason codes can be retrieved using **getLastErrorStatus**, even if AmWarningExceptions are disabled.

```
void  enableWarnings(boolean warningsOn);
```

`warningsOn`	If set to `true`, AmWarningExceptions will be raised for this object.

getCCSID

Returns the coded character set identifier for the AmSender. A non-default value reflects the CCSID of a remote system unable to perform CCSID conversion of received messages. In this case the sender must perform CCSID conversion of the message before it is sent.

```
int  getCCSID();
```

getEncoding

Returns the value used to encode numeric data types for the AmSender. A non-default value reflects the encoding of a remote system unable to convert the encoding of received messages. In this case the sender must convert the encoding of the message before it is sent.

```
int  getEncoding();
```

getLastErrorStatus

Returns the AmStatus of the last error condition.

```
AmStatus  getLastErrorStatus();
```

getName

Returns the name of the AmSender.

```
String  getName();
```

open

Opens an AmSender service. **open** is overloaded: the `policy` parameter is optional.

```
void  open(AmPolicy policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

send

Sends a message to the destination specified by the AmSender. If the AmSender is not open, it will be opened (if this action is specified in the policy options).

send is overloaded: the `sendMessage` parameter is required, but the others are optional. `receivedMessage` and `responseService` are used in request/response messaging, and are mutually exclusive.

```
void  send(
  AmMessage   sendMessage,
  AmReceiver  responseService,
  AmMessage   receivedMessage,
  AmPolicy    policy);
```

`sendMessage`	The message object that contains the data to be sent.
`responseService`	The AmReceiver to be used for receiving any response to the sent message. If omitted, no response can be received.
`receivedMessage`	The previously received message which is used for correlation with the sent message. If omitted, the sent message is not correlated with any received message.
`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

AmReceiver

An **AmReceiver** object encapsulates an MQSeries object descriptor (MQOD) structure. This represents an MQSeries queue on a local or remote queue manager. An open AmReceiver is always associated with an open connection object, such as a queue manager connection. Support is also included for a dynamic AmReceiver (that encapsulates a model queue). The required AmReceiver object definitions can be provided from a repository or can be created automatically from the set of existing queue objects available on the local queue manager.

There is a definition type associated with each AmReceiver:

```
AMDT_UNDEFINED
AMDT_TEMP_DYNAMIC
AMDT_DYNAMIC
AMDT_PREDEFINED
```

An AmReceiver created from a repository definition will be initially of type AMDT_PREDEFINED or AMDT_DYNAMIC. When opened, its definition type might change from AMDT_DYNAMIC to AMDT_TEMP_DYNAMIC according to the properties of its underlying queue object.

An AmReceiver created with default values (that is, without a repository definition) will have its definition type set to AMDT_UNDEFINED until it is opened. When opened, this will become AMDT_DYNAMIC, AMDT_TEMP_DYNAMIC, or AMDT_PREDEFINED, according to the properties of its underlying queue object.

browse

Browses an AmReceiver service. **browse** is overloaded: the `browseMessage` and `options` parameters are required, but the others are optional.

```
void  browse(
  AmMessage   browseMessage,
  int         options,
  AmSender    responseService,
  AmPolicy    policy);
```

`browseMessage` The message object that receives the browse data.

`options` Options controlling the browse operation. Possible values are:

```
AMBRW_NEXT
AMBRW_FIRST
AMBRW_CURRENT
AMBRW_RECEIVE_CURRENT
AMBRW_DEFAULT        (AMBRW_NEXT)
AMBRW_LOCK_NEXT      (AMBRW_LOCK + AMBRW_NEXT)
AMBRW_LOCK_FIRST     (AMBRW_LOCK + AMBRW_FIRST)
AMBRW_LOCK_CURRENT   (AMBRW_LOCK + AMBRW_CURRENT)
AMBRW_UNLOCK
```

`AMBRW_RECEIVE_CURRENT` is equivalent to **AmReceiver.receive** for the message under the browse cursor.

Note that a locked message is unlocked by another browse or receive, even though it is not for the same message.

`responseService`	The AmSender to be used for sending any response to the browsed message. If omitted, no response can be sent.
`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

clearErrorCodes

Clears the error codes in the AmReceiver.

```
  void  clearErrorCodes();
```

close

Closes the AmReceiver. **close** is overloaded: the `policy` parameter is optional.

```
  void  close(AmPolicy policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

enableWarnings

Enables AmWarningExceptions; the default value for any AmObject is that AmWarningExceptions are not raised. Note that warning reason codes can be retrieved using **getLastErrorStatus**, even if AmWarningExceptions are disabled.

```
  void  enableWarnings(boolean warningsOn);
```

`warningsOn`	If set to `true`, AmWarningExceptions will be raised for this object.

getDefinitionType

Returns the definition type (service type) for the AmReceiver.

```
  int  getDefinitionType();
```

The following values can be returned:

```
AMDT_UNDEFINED
AMDT_TEMP_DYNAMIC
AMDT_DYNAMIC
AMDT_PREDEFINED
```

Values other than AMDT_UNDEFINED reflect the properties of the underlying queue object.

getLastErrorStatus

Returns the AmStatus of the last error condition.

```
  AmStatus  getLastErrorStatus();
```

getName

Returns the name of the AmReceiver.

```
String  getName();
```

getQueueName

Returns the queue name of the AmReceiver. This is used to determine the queue name of a permanent dynamic AmReceiver, so that it can be recreated with the same queue name in order to receive messages in a subsequent session. (See also **setQueueName**.)

```
String  getQueueName();
```

open

Opens an AmReceiver service. **open** is overloaded: the `policy` parameter is optional.

```
void  open(AmPolicy policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

receive

Receives a message from the AmReceiver service. **receive** is overloaded: the `receiveMessage` parameter is required, but the others are optional.

```
void  receive(
  AmMessage   receiveMessage,
  AmSender    responseService,
  AmMessage   selectionMessage,
  AmPolicy    policy);
```

`receiveMessage`	The message object that receives the data. The message object is reset implicitly before the receive takes place.
`responseService`	The AmSender to be used for sending any response to the received message. If omitted, no response can be sent.
`selectionMessage`	A message object which contains the correlation ID used to selectively receive a message from the AmReceiver. If omitted, the first available message is received.
`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

setQueueName

Sets the queue name of the AmReceiver (when this encapsulates a model queue). This is used to specify the queue name of a recreated permanent dynamic AmReceiver, in order to receive messages in a session subsequent to the one in which it was created. (See also **getQueueName**.)

```
void  setQueueName(String queueName);
```

`queueName`	The queue name to be set in the AmReceiver.

AmDistributionList

An **AmDistributionList** object encapsulates a list of AmSender objects.

clearErrorCodes

Clears the error codes in the AmDistributionList.

```
void  clearErrorCodes();
```

close

Closes the AmDistributionList. **close** is overloaded: the `policy` parameter is optional.

```
void  close(AmPolicy policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

enableWarnings

Enables AmWarningExceptions; the default value for any AmObject is that AmWarningExceptions are not raised. Note that warning reason codes can be retrieved using **getLastErrorStatus**, even if AmWarningExceptions are disabled.

```
void  enableWarnings(boolean warningsOn);
```

`warningsOn`	If set to `true`, AmWarningExceptions will be raised for this object.

getLastErrorStatus

Returns the AmStatus of the last error condition of this object.

```
AmStatus  getLastErrorStatus();
```

getName

Returns the name of the AmDistributionList object.

```
String  getName();
```

getSender

Returns the AmSender in the AmDistributionList object at the index specified. `AmDistributionList.getSenderCount` gets the number of AmSender services in the distribution list.

```
AmSender  getSender(int index);
```

`index`	The index of the AmSender in the AmDistributionList, starting at zero.

getSenderCount

Returns the number of AmSender services in the AmDistributionList object.

```
int  getSenderCount();
```

open

Opens an AmDistributionList object for each of the destinations in the distribution list. **open** is overloaded: the `policy` parameter is optional.

```
void  open(AmPolicy policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

send

Sends a message to each AmSender defined in the AmDistributionList object. **send** is overloaded: the `sendMessage` parameter is required, but the others are optional.

```
void  send(
  AmMessage   sendMessage,
  AmReceiver  responseService,
  AmPolicy    policy);
```

`sendMessage`	The message object containing the data to be sent.
`responseService`	The AmReceiver to be used for receiving any response to the sent message. If omitted, no response can be received.
`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

AmPublisher

An **AmPublisher** object encapsulates an AmSender and provides support for publish requests to a publish/subscribe broker.

clearErrorCodes

Clears the error codes in the AmPublisher.

```
void  clearErrorCodes();
```

close

Closes the AmPublisher. **close** is overloaded: the `policy` parameter is optional.

```
void  close(AmPolicy policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

enableWarnings

Enables AmWarningExceptions; the default value for any AmObject is that AmWarningExceptions are not raised. Note that warning reason codes can be retrieved using **getLastErrorStatus**, even if AmWarningExceptions are disabled.

```
void  enableWarnings(boolean warningsOn);
```

`warningsOn`	If set to `true`, AmWarningExceptions will be raised for this object.

getCCSID

Returns the coded character set identifier for the AmPublisher. A non-default value reflects the CCSID of a remote system unable to perform CCSID conversion of received messages. In this case the publisher must perform CCSID conversion of the message before it is sent.

```
int  getCCSID();
```

getEncoding

Returns the value used to encode numeric data types for the AmPublisher. A non-default value reflects the encoding of a remote system unable to convert the encoding of received messages. In this case the publisher must convert the encoding of the message before it is sent.

```
int  getEncoding();
```

getLastErrorStatus

Returns the AmStatus of the last error condition.

```
AmStatus  getLastErrorStatus();
```

getName

Returns the name of the AmPublisher.

```
String  getName();
```

open

Opens an AmPublisher service. **open** is overloaded: the `policy` parameter is optional.

```
void  open(AmPolicy policy);
```

`policy`	The policy to be used. If omitted, the system default policy (AMSD_POL) is used.

publish

Publishes a message using the AmPublisher. **publish** is overloaded: the `pubMessage` parameter is required, but the others are optional.

```
void  publish(
  AmMessage   pubMessage,
  AmReceiver  responseService,
  AmPolicy    policy);
```

`pubMessage`	The message object that contains the data to be published.
`responseService`	The AmReceiver to which the response to the publish request should be sent. Omit it if no response is required. This parameter is mandatory if the policy specifies implicit registration of the publisher.
`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

AmSubscriber

An **AmSubscriber** object encapsulates both an AmSender and an AmReceiver. It provides support for subscribe and unsubscribe requests to a publish/subscribe broker, and for receiving publications from the broker.

clearErrorCodes

Clears the error codes in the AmSubscriber.

```
void  clearErrorCodes();
```

close

Closes the AmSubscriber. **close** is overloaded: the `policy` parameter is optional.

```
void  close(AmPolicy policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

enableWarnings

Enables AmWarningExceptions; the default value for any AmObject is that AmWarningExceptions are not raised. Note that warning reason codes can be retrieved using **getLastErrorStatus**, even if AmWarningExceptions are disabled.

```
void  enableWarnings(boolean warningsOn);
```

`warningsOn`	If set to `true`, AmWarningExceptions will be raised for this object.

getCCSID

Returns the coded character set identifier for the AmSender in the AmSubscriber. A non-default value reflects the CCSID of a remote system unable to perform CCSID conversion of received messages. In this case the subscriber must perform CCSID conversion of the message before it is sent.

```
int  getCCSID();
```

getDefinitionType

Returns the definition type for the AmReceiver in the AmSubscriber.

```
int  getDefinitionType();
```

The following values can be returned:

```
AMDT_UNDEFINED
AMDT_TEMP_DYNAMIC
AMDT_DYNAMIC
AMDT_PREDEFINED
```

getEncoding

Returns the value used to encode numeric data types for the AmSender in the AmSubscriber. A non-default value reflects the encoding of a remote system unable to convert the encoding of received messages. In this case the subscriber must convert the encoding of the message before it is sent.

```
int  getEncoding();
```

getLastErrorStatus

Returns the AmStatus of the last error condition.

```
AmStatus  getLastErrorStatus();
```

getName

Returns the name of the AmSubscriber.

```
String  getName();
```

getQueueName

Returns the queue name used by the AmSubscriber to receive messages. This is used to determine the queue name of a permanent dynamic AmReceiver in the AmSubscriber, so that it can be recreated with the same queue name in order to receive messages in a subsequent session. (See also **setQueueName**.)

```
String  getQueueName();
```

open

Opens an AmSubscriber. **open** is overloaded: the `policy` parameter is optional.

```
void  open(AmPolicy policy);
```

`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

receive

Receives a message, normally a publication, using the AmSubscriber. The message data, topic and other elements can be accessed using the message interface methods (see page 228).

receive is overloaded: the `pubMessage` parameter is required, but the others are optional.

```
void  receive(
  AmMessage   pubMessage,
  AmMessage   selectionMessage,
  AmPolicy    policy);
```

`pubMessage`	The message object containing the data that has been published. The message object is reset implicitly before the receive takes place.
`selectionMessage`	A message object containing the correlation ID used to selectively receive a message from the AmSubscriber. If omitted, the first available message is received.
`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

setQueueName

Sets the queue name in the AmReceiver of the AmSubscriber, when this encapsulates a model queue. This is used to specify the queue name of a recreated permanent dynamic AmReceiver, in order to receive messages in a session subsequent to the one in which it was created. (See also **setQueueName**.)

```
void  setQueueName(String queueName);
```

`queueName`	The queue name to be set.

subscribe

Sends a subscribe message to a publish/subscribe broker using the AmSubscriber, to register a subscription. The topic and other elements can be specified using the message interface methods (see page 228) before sending the message.

Publications matching the subscription are sent to the AmReceiver associated with the AmSubscriber. By default, this has the same name as the AmSubscriber, with the addition of the suffix '.RECEIVER'.

subscribe is overloaded: the `subMessage` parameter is required, but the others are optional.

```
void  subscribe(
  AmMessage   subMessage,
  AmReceiver  responseService,
  AmPolicy    policy);
```

`subMessage`	The message object that contains the topic subscription data.
`responseService`	The AmReceiver to which the response to this subscribe request should be sent. Omit it if no response is required. This is not the AmReceiver to which publications will be sent by the broker; they are sent to the AmReceiver associated with the AmSubscriber (see above).
`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

unsubscribe

Sends an unsubscribe message to a publish/subscribe broker using the AmSubscriber, to deregister a subscription. The topic and other elements can be specified using the message interface methods (see page 228) before sending the message.

unsubscribe is overloaded: the `unsubMessage` parameter is required, but the others are optional.

```
void  unsubscribe(
  AmMessage   unsubMessage,
  AmReceiver  responseService,
  AmPolicy    policy);
```

`unsubMessage`	The message object that contains the topics to which the unsubscribe request applies.
`responseService`	The AmReceiver to which the response to this unsubscribe request should be sent. Omit it if no response is required.
`policy`	The policy to be used. If omitted, the system default policy (constant: AMSD_POL) is used.

AmPolicy

An **AmPolicy** object encapsulates details of how the AMI processes the message (for instance, the priority and persistence of the message, how errors are handled, and whether transactional processing is used).

clearErrorCodes

Clears the error codes in the AmPolicy.

```
void  clearErrorCodes();
```

enableWarnings

Enables AmWarningExceptions; the default value for any AmObject is that AmWarningExceptions are not raised. Note that warning reason codes can be retrieved using **getLastErrorStatus**, even if AmWarningExceptions are disabled.

```
void  enableWarnings(boolean warningsOn);
```

`warningsOn`	If set to `true`, AmWarningExceptions will be raised for this object.

getLastErrorStatus

Returns the AmStatus of the last error condition.

```
AmStatus  getLastErrorStatus();
```

getName

Returns the name of the AmPolicy object.

```
String  getName();
```

getWaitTime

Returns the wait time (in ms) set for this AmPolicy.

```
int  getWaitTime();
```

setWaitTime

Sets the wait time for any **receive** using this AmPolicy.

```
void  setWaitTime(int waitTime);
```

`waitTime`	The wait time (in ms) to be set in the AmPolicy.

AmConstants

This class provides access to the AMI constants listed in Appendix B, “Constants” on page 313.

For example, to use the constant AMRC_NONE (an AMI reason code), specify AmConstants.AMRC_NONE.

Note: Not all of the constants available in the C and C++ programming interfaces are available in Java, because they are not all appropriate in this language. For instance, AmConstants does not contain AMB_TRUE or AMB_FALSE, since the Java language has its own true and false constants and these are used by the AMI for Java.

AmElement

An **AmElement** object encapsulates a name/value pair which can be added to an AmMessage object.

AmElement

Constructor for an AmElement object.

```
AmElement(String name, String value);
```

`name`	The name of the element.
`value`	The value of the element.

getName

Returns the name of the AmElement.

```
String  getName();
```

getValue

Returns the value of the AmElement.

```
String  getValue();
```

getVersion

Returns the version of the AmElement (the default value is AmConstants.AMELEM_VERSION_1).

```
int  getVersion();
```

setVersion

Sets the version of the AmElement.

```
void  setVersion(int version);
```

`version`	The version of the AmElement that is set. It can take the value AmConstants.AMELEM_VERSION_1 or AmConstants.AMELEM_CURRENT_VERSION.

toString

Returns a String representation of the AmElement.

```
String  toString();
```

AmObject

AmObject is a Java Interface. The following classes implement the AmObject interface:

AmSession
AmMessage
AmSender
AmReceiver
AmDistributionList
AmPublisher
AmSubscriber
AmPolicy

This allows application programmers to use generic error handling routines.

clearErrorCodes

Clears the error codes in the AmObject.

```
void  clearErrorCodes();
```

getLastErrorStatus

Returns the AmStatus of the last error condition.

```
AmStatus  getLastErrorStatus();
```

getName

Returns the name of the AmObject.

```
String  getName();
```

AmStatus

An **AmStatus** object encapsulates the error status of other AmObjects.

AmStatus

Constructor for an AmStatus object.

```
AmStatus();
```

getCompletionCode

Returns the completion code from the AmStatus object.

```
int  getCompletionCode();
```

getReasonCode

Returns the reason code from the AmStatus object.

```
int  getReasonCode();
```

getReasonCode2

Returns the secondary reason code from the AmStatus object. (This code is specific to the underlying transport used by the AMI). For MQSeries, the secondary reason code is an MQSeries reason code of type MQRC_xxx.

```
int  getReasonCode2();
```

toString

Returns a String representation of the internal state of the AmStatus object.

```
String  toString();
```

AmException

AmException is the base Exception class; all other Exceptions inherit from this class.

getClassName

Returns the type of object throwing the Exception.

```
String  getClassName();
```

getCompletionCode

Returns the completion code for the Exception.

```
int  getCompletionCode();
```

getMethodName

Returns the name of the method throwing the Exception.

```
String  getMethodName();
```

getReasonCode

Returns the reason code for the Exception.

```
int  getReasonCode();
```

getSource

Returns the AmObject throwing the Exception.

```
AmObject  getSource();
```

toString

Returns a String representation of the Exception.

```
String  toString();
```

AmErrorException

An Exception of type **AmErrorException** is raised when an object experiences an error with a severity level of FAILED (CompletionCode = AMCC_FAILED).

getClassName

Returns the type of object throwing the Exception.

```
String  getClassName();
```

getCompletionCode

Returns the completion code for the Exception.

```
int  getCompletionCode();
```

getMethodName

Returns the name of the method throwing the Exception.

```
String  getMethodName();
```

getReasonCode

Returns the reason code for the Exception.

```
int  getReasonCode();
```

getSource

Returns the AmObject throwing the Exception.

```
AmObject  getSource();
```

toString

Returns a String representation of the Exception.

```
String  toString();
```

AmWarningException

An Exception of type **AmWarningException** is raised when an object experiences an error with a severity level of WARNING (CompletionCode = AMCC_WARNING).

getClassName

Returns the type of object throwing the Exception.

```
String  getClassName();
```

getCompletionCode

Returns the completion code for the Exception.

```
int  getCompletionCode();
```

getMethodName

Returns the name of the method throwing the Exception.

```
String  getMethodName();
```

getReasonCode

Returns the reason code for the Exception.

```
int  getReasonCode();
```

getSource

Returns the AmObject throwing the Exception.

```
AmObject  getSource();
```

toString

Returns a String representation of the Exception.

```
String  toString();
```

Part 5. Setting up an AMI installation

This part contains:

Chapter 12. Installation and sample programs

The Application Messaging Interface is available for the AIX, Sun Solaris, Windows NT and Windows 98 platforms.

This chapter contains:

Prerequisites

Prior to installing the AMI you should make sure that your system has sufficient disk space, and the software listed below.

Disk space

Disk space requirements:

AIX	11.6 MB
Sun Solaris	6.7 MB
Windows	6.6 MB

Operating environments

The AMI runs under the following operating systems:

AIX	V4.2 and V4.3
Sun Solaris	V2.6 and V2.7
Windows	Windows NT V4 and Windows 98

MQSeries environment

You can run the AMI in an MQSeries server or client environment.

To run the AMI in an MQSeries server environment you need at least one of the following installed on your system:

- MQSeries for AIX Version 5.1 or later
- MQSeries for Sun Solaris Version 5.1 or later
- MQSeries for Windows NT Version 5.1 or later

To run the AMI in an MQSeries client environment you need at least one of the following installed on your system:

- MQSeries client for AIX Version 5.1 or later
- MQSeries client for Sun Solaris Version 5.1 or later
- MQSeries client for Windows NT Version 5.1 or later
- MQSeries client for Windows 98 Version 5.1 or later

The MQSeries client requires access to at least one supporting MQSeries server.

Language compilers

The following language compilers for C, C++ and Java are supported:

AIX	C Set ++ 3.1.4.7 and above JDK 1.1.7 and above
Sun Solaris	Workshop Compiler 4.2 JDK 1.1.7 and above
Windows	Microsoft Visual C++ 6 JDK 1.1.7 and above

Next step

Now go to one of the following to start the installation procedure:

- “Installation on AIX” on page 261
- “Installation on Sun Solaris” on page 265
- “Installation on Windows” on page 269

Installation on AIX

The AMI package for AIX comes as a compressed archive file, `ma0f_ax.tar.Z`. Uncompress and restore it as follows:

1. Login as `root`
2. Store `ma0f_ax.tar.Z` in `/tmp`
3. Execute `uncompress -fv /tmp/ma0f_ax.tar.Z`
4. Execute `tar -xvf /tmp/ma0f_ax.tar`
5. Execute `rm /tmp/ma0f_ax.tar`

This creates the following files:

amt100.tar	A standard tar file containing the AMI files
amtInstall	A script file to aid AMI installation
amtRemove	A script file to aid AMI removal
readme	A file containing any product and information updates that have become available since this documentation was produced

Installation

Installation can be carried out manually, or using the **amtInstall** utility.

Manual installation

Restore the tar file `amt100.tar`. This should be done under the base MQSeries directory `/usr/mqm`, so that the AMI tar file restores to a directory structure consistent with MQSeries. This operation usually requires root access. Existing files will be overwritten. (Note that the location `/usr/mqm/` is consistent with MQSeries Version 5.1, which is the prerequisite for the AMI).

Using amtInstall

1. Login as `root`
2. Execute `amtInstall <directory>`

where `<directory>` is the directory containing the `amt100.tar` file.

The **amtInstall** utility will unpack the tar file into the correct location and provide the necessary links for your environment. Existing files will be overwritten.

Note: All files and directories created must be accessible to all AMI users. These files are listed in “Directory structure (AIX)” on page 263.

Removing the AMI

Run the **amtRemove** utility to remove all the files that were created by **amtInstall**.

Setting the runtime environment

Make sure the location of the AMI runtime binary files is added to your PATH environment variable. For example:

```
export PATH=$PATH:/usr/mqm/lib:
```

Note: The above step is not needed if you used the **amtInstall** utility.

In addition, for the samples:

```
export PATH=$PATH:/usr/mqm/amt/samp/C/bin:/usr/mqm/amt/samp/Cpp/bin:
```

Java programs

When running Java, there are some additional steps.

The AMI classes must be contained in the CLASSPATH, for example:

```
export CLASSPATH=$CLASSPATH:/usr/mqm/java/lib/com.ibm.mq.amt.jar:
```

In addition, for the samples:

```
export CLASSPATH=$CLASSPATH:/usr/mqm/amt/samp/java/bin
                   /com.ibm.mq.amt.samples.jar:
```

Also, in order to load the AMI library for Java:

```
export LIBPATH=$LIBPATH:/usr/mqm/lib:
```

Next step

Now go to “Local host and repository files” on page 272 to continue the installation procedure.

Directory structure (AIX)

The AMI tar file contains:

```
/amt/amtsdfts.tst : MQSeries mqsc command file to create default MQSeries
  objects required by the AMI

/amt/amthost.xml : Sample AMI XML file used as the default host file

/amt/amt.dtd : AMI Document Type Definition file on which the AMI
  repository is based

/amt/inc
   amtc.h : The C header file for the AMI
   amtcpp.hpp : The C++ header file for the AMI

/amt/intlFiles/locales : Directory containing data translation files

/amt/ipla : The International Program License Agreement file
/amt/li : The License Information file

/java/lib
   com.ibm.mq.amt.jar : The jar file containing the AMI classes for Java

/lib
   libamt.a : The main AMI library
   libamt_r.a : The main AMI threaded library
   libamtXML.a : The AMI XML parsing library
   libamtXML_r.a : The AMI threaded XML parsing library
   libamtCpp.a : The AMI C++ library
   libamtCpp_r.a : The AMI C++ threaded library
   libamtJava.so: The AMI JNI library
   libamtICUUC.a : The AMI codepage translation library
   libamtICUUC_r.a : The AMI codepage translation threaded library
   amtcmqm : Dynamic binding stub for MQSeries Server library
   amtcmqm_r : Dynamic binding stub for MQSeries Server threaded library
   amtcmqic : Dynamic binding stub for MQSeries Client library
   amtcmqic_r : Dynamic binding stub for MQSeries Client threaded library

/amt/samp
   amtsamp.tst : MQSeries mqsc command file to create MQSeries objects
     required by AMI samples
   amt.xml : Sample AMI XML repository for use with the AMI samples
```

```
/amt/samp/C
   amtsosnd.c : C source for object-level send and forget sample
   amtsorcv.c : C source for object-level receiver sample
   amtsoclt.c : C source for object-level client sample
   amtsosvr.c : C source for object-level server sample
   amtsopub.c : C source for object-level publisher sample
   amtsosub.c : C source for object-level subscriber sample
   amtshsnd.c : C source for high-level send and forget sample
   amtshrcv.c : C source for high-level receiver sample
   amtshclt.c : C source for high-level client sample
   amtshsvr.c : C source for high-level server sample
   amtshpub.c : C source for high-level publisher sample
   amtshsub.c : C source for high-level subscriber sample

/amt/samp/C/bin
   amtsosnd : C object-level send and forget sample program
   amtsorcv : C object-level receiver sample program
   amtsoclt : C object-level client sample program
   amtsosvr : C object-level server sample program
   amtsopub : C object-level publisher sample program
   amtsosub : C object-level subscriber sample program
   amtshsnd : C high-level send and forget sample program
   amtshrcv : C high-level receiver sample program
   amtshclt : C high-level client sample program
   amtshsvr : C high-level server sample program
   amtshpub : C high-level publisher sample program
   amtshsub : C high-level subscriber sample program

/amt/samp/Cpp
   SendAndForget.cpp : C++ source for send and forget sample
   Receiver.cpp : C++ source for receiver sample
   Client.cpp : C++ source for client sample
   Server.cpp : C++ source for server sample
   Publisher.cpp : C++ source for publisher sample
   Subscriber.cpp : C++ source for subscriber sample

/amt/samp/Cpp/bin
   SendAndForget : C++ send and forget sample program
   Receiver : C++ receiver sample program
   Client : C++ client sample program
   Server : C++ server sample program
   Publisher : C++ publisher sample program
   Subscriber : C++ subscriber sample program

/amt/samp/java
   SendAndForget.java : Java source for send and forget sample
   Receiver.java : Java source for receiver sample
   Client.java : Java source for client sample
   Server.java : Java source for server sample
   Publisher.java : Java source for publisher sample
   Subscriber.java : Java source for subscriber sample

/amt/samp/java/bin
   com.ibm.mq.amt.samples.jar : The jar file containing the AMI
     samples class files for Java
```

Installation on Sun Solaris

The AMI package for Sun Solaris comes as a compressed archive file, `ma0f_sol.tar.Z`. Uncompress and restore it as follows:

1. Login as `root`
2. Store `ma0f_sol.tar.Z` in `/tmp`
3. Execute `uncompress -fv /tmp/ma0f_sol.tar.Z`
4. Execute `tar -xvf /tmp/ma0f_sol.tar`
5. Execute `rm /tmp/ma0f_sol.tar`

This creates the following files:

amt100.tar	A standard tar file containing the AMI files
amtInstall	A script file to aid AMI installation
amtRemove	A script file to aid AMI removal
readme	A file containing any product and information updates that have become available since this documentation was produced

Installation

Installation can be carried out manually, or using the **amtInstall** utility.

Manual installation

Restore the tar file `amt100.tar`. This should be done under the base MQSeries directory `/opt/mqm`, so that the AMI tar file restores to a directory structure consistent with MQSeries. This operation usually requires root access. Existing files will be overwritten.

Using amtInstall

1. Login as `root`
2. Execute `amtInstall <directory>`

where `<directory>` is the directory containing the `amt100.tar` file.

The **amtInstall** utility will unpack the tar file into the correct location and provide the necessary links for your environment. Existing files will be overwritten.

Note: All files and directories created must be accessible to all AMI users. These files are listed in “Directory structure (Solaris)” on page 267.

Removing the AMI

Run the **amtRemove** utility to remove all the files that were created by **amtInstall**.

Setting the runtime environment

Make sure the location of the AMI runtime binary files is added to your PATH environment variable. For example:

```
export PATH=$PATH:/opt/mqm/lib:
```

Note: The above step is not needed if you used the **amtInstall** utility.

In addition, for the samples:

```
export PATH=$PATH:/opt/mqm/amt/samp/C/bin:/opt/mqm/amt/samp/Cpp/bin:
```

Java programs

When running Java, there are some additional steps.

The AMI classes must be contained in the CLASSPATH, for example:

```
export CLASSPATH=$CLASSPATH:/opt/mqm/java/lib/com.ibm.mq.amt.jar:
```

In addition, for the samples:

```
export CLASSPATH=$CLASSPATH:/opt/mqm/amt/samp/java/bin
                   /com.ibm.mq.amt.samples.jar:
```

Also, in order to load the AMI library for Java:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/opt/mqm/lib:
```

Next step

Now go to “Local host and repository files” on page 272 to continue the installation procedure.

Directory structure (Solaris)

The AMI tar file contains:

```
/amt/amtsdfts.tst : MQSeries mqsc command file to create default MQSeries
  objects required by the AMI

/amt/amthost.xml : Sample AMI XML file used as the default host file

/amt/amt.dtd : AMI Document Type Definition file on which the AMI
  repository is based

/amt/inc
   amtc.h : The C header file for the AMI
   amtcpp.hpp : The C++ header file for the AMI

/amt/intlFiles/locales : Directory containing data translation files

/amt/ipla : The International Program License Agreement file
/amt/li : The License Information file

/java/lib
   com.ibm.mq.amt.jar : The jar file containing the AMI classes for Java

/lib
   libamt.so : The main AMI library
   libamtXML.so : The AMI XML parsing library
   libamtCpp.so : The AMI C++ library
   libamtJava.so: The AMI JNI library
   libamtICUUC.so : The AMI codepage translation library
   amtcmqm : Dynamic binding stub for MQSeries Server library
   amtcmqic : Dynamic binding stub for MQSeries Client library

/amt/samp
   amtsamp.tst : MQSeries mqsc command file to create MQSeries objects
     required by AMI samples
   amt.xml : Sample AMI XML repository for use with the AMI samples
```

```
/amt/samp/C
   amtsosnd.c : C source for object-level send and forget sample
   amtsorcv.c : C source for object-level receiver sample
   amtsoclt.c : C source for object-level client sample
   amtsosvr.c : C source for object-level server sample
   amtsopub.c : C source for object-level publisher sample
   amtsosub.c : C source for object-level subscriber sample
   amtshsnd.c : C source for high-level send and forget sample
   amtshrcv.c : C source for high-level receiver sample
   amtshclt.c : C source for high-level client sample
   amtshsvr.c : C source for high-level server sample
   amtshpub.c : C source for high-level publisher sample
   amtshsub.c : C source for high-level subscriber sample

/amt/samp/C/bin
   amtsosnd : C object-level send and forget sample program
   amtsorcv : C object-level receiver sample program
   amtsoclt : C object-level client sample program
   amtsosvr : C object-level server sample program
   amtsopub : C object-level publisher sample program
   amtsosub : C object-level subscriber sample program
   amtshsnd : C high-level send and forget sample program
   amtshrcv : C high-level receiver sample program
   amtshclt : C high-level client sample program
   amtshsvr : C high-level server sample program
   amtshpub : C high-level publisher sample program
   amtshsub : C high-level subscriber sample program

/amt/samp/Cpp
   SendAndForget.cpp : C++ source for send and forget sample
   Receiver.cpp : C++ source for receiver sample
   Client.cpp : C++ source for client sample
   Server.cpp : C++ source for server sample
   Publisher.cpp : C++ source for publisher sample
   Subscriber.cpp : C++ source for subscriber sample

/amt/samp/Cpp/bin
   SendAndForget : C++ send and forget sample program
   Receiver : C++ receiver sample program
   Client : C++ client sample program
   Server : C++ server sample program
   Publisher : C++ publisher sample program
   Subscriber : C++ subscriber sample program

/amt/samp/java
   SendAndForget.java : Java source for send and forget sample
   Receiver.java : Java source for receiver sample
   Client.java : Java source for client sample
   Server.java : Java source for server sample
   Publisher.java : Java source for publisher sample
   Subscriber.java : Java source for subscriber sample

/amt/samp/java/bin
   com.ibm.mq.amt.samples.jar : The jar file containing the AMI
     samples class files for Java
```

Installation on Windows

The AMI package for Windows 98 and Windows NT comes as a zip file, `ma0f_nt.zip`. Once unzipped it comprises:

readme	A file containing any product and information updates that have become available since this documentation was produced
setup	InstallShield installation program for MQSeries AMI

In addition, it contains files used by the **setup** program.

Installation

1. Create an empty directory called `tmp` and make it current.
2. Store the `ma0f_nt.zip` file in this directory.
3. Uncompress it into `tmp` using Info-ZIP's UnZip program (or other unzip program).
4. Run **setup**.
5. Delete the `tmp` directory.

The files and directories created are listed in "Directory structure (Windows)" on page 270.

Removing the AMI

To uninstall the Application Messaging Interface, use the Add/Remove Programs control panel.

Note: You **must** remove the AMI entries from the CLASSPATH (for instance, C:\MQSeries\java\lib\com.ibm.mq.amt.jar; and C:\MQSeries\amt\samples\java\bin\com.ibm.mq.amt.samples.jar;). These will not be removed by Add/Remove Programs.

In addition, if you specified a directory other than the default during installation, you must remove this directory from the PATH environment variable.

Setting the runtime environment

By default, the location of the AMI runtime binary files matches that of MQSeries (for example `C:\MQSeries\bin`) and is added to the PATH environment variable by the **setup** program.

If you specified a different directory for the runtime files, you **must** add it to the PATH environment variable yourself. (See also "Removing the AMI.")

To use the samples, add the sample C and C++ binary directories to your PATH environment variable. For example (assuming that the root directory for MQSeries is `C:\MQSeries`):

```
set PATH=%PATH%;C:\MQSeries\amt\samples\C\bin;
           C:\MQSeries\amt\samples\Cpp\bin;
```

When running Java, the AMI classes (`C:\MQSeries\java\lib\com.ibm.mq.amt.jar`) and samples (`C:\MQSeries\amt\samples\java\bin\com.ibm.mq.amt.samples.jar`) must be contained in the CLASSPATH environment variable. This is done by the **setup** program.

Next step

Now go to “Local host and repository files” on page 272 to continue the installation procedure.

Directory structure (Windows)

On Windows platforms the directory structure contains:

```
\amt\amtsdfts.tst : MQSeries mqsc command file to create default MQSeries
  objects required by the AMI

\amt\amthost.xml : Sample AMI XML file used as the default host file

\amt\amt.dtd : AMI Document Type Definition file on which the AMI
  repository is based

\amt\include
   amtc.h : The C header file for the AMI
   amtcpp.hpp : The C++ header file for the AMI

\amt\intlFiles\locales : Directory containing data translation files

\amt\ipla : The International Program License Agreement file
\amt\li : The License Information file

\java\lib
   com.ibm.mq.amt.jar : The jar file containing the AMI classes for Java

\bin
   amt.dll : The main AMI library
   amt.lib : The AMI LIB file used for building C programs
   amtXML.dll : The AMI XML parsing library
   amtCpp.dll : The AMI C++ library
   amtCpp.lib : The AMI LIB file used for building C++ programs
   amtJava.dll: The AMI JNI library
   amtICUUC.dll : The AMI codepage translation library

   MSVCRT.DLL : Main MVSC runtime library
   MSVCIRT.DLL : Iostream MSVC runtime library

\amt\samples
   amtsamp.tst : MQSeries mqsc command file to create MQSeries objects
     required by AMI samples
   amt.xml : Sample AMI XML repository for use with the AMI samples
```

```
\amt\samples\C
   amtsosnd.c : C source for object-level send and forget sample
   amtsorcv.c : C source for object-level receiver sample
   amtsoclt.c : C source for object-level client sample
   amtsosvr.c : C source for object-level server sample
   amtsopub.c : C source for object-level publisher sample
   amtsosub.c : C source for object-level subscriber sample
   amtshsnd.c : C source for high-level send and forget sample
   amtshrcv.c : C source for high-level receiver sample
   amtshclt.c : C source for high-level client sample
   amtshsvr.c : C source for high-level server sample
   amtshpub.c : C source for high-level publisher sample
   amtshsub.c : C source for high-level subscriber sample

\amt\samples\C\bin
   amtsosnd.exe : C object-level send and forget sample program
   amtsorcv.exe : C object-level receiver sample program
   amtsoclt.exe : C object-level client sample program
   amtsosvr.exe : C object-level server sample program
   amtsopub.exe : C object-level publisher sample program
   amtsosub.exe : C object-level subscriber sample program
   amtshsnd.exe : C high-level send and forget sample program
   amtshrcv.exe : C high-level receiver sample program
   amtshclt.exe : C high-level client sample program
   amtshsvr.exe : C high-level server sample program
   amtshpub.exe : C high-level publisher sample program
   amtshsub.exe : C high-level subscriber sample program

\amt\samples\Cpp
   SendAndForget.cpp : C++ source for send and forget sample
   Receiver.cpp : C++ source for receiver sample
   Client.cpp : C++ source for client sample
   Server.cpp : C++ source for server sample
   Publisher.cpp : C++ source for publisher sample
   Subscriber.cpp : C++ source for subscriber sample

\amt\samples\Cpp\bin
   SendAndForget.exe : C++ send and forget sample program
   Receiver.exe : C++ receiver sample program
   Client.exe : C++ client sample program
   Server.exe : C++ server sample program
   Publisher.exe : C++ publisher sample program
   Subscriber.exe : C++ subscriber sample program

\amt\samples\java
   SendAndForget.java : Java source for send and forget sample
   Receiver.java : Java source for receiver sample
   Client.java : Java source for client sample
   Server.java : Java source for server sample
   Publisher.java : Java source for publisher sample
   Subscriber.java : Java source for subscriber sample

\amt\samples\java\bin
   com.ibm.mq.amt.samples.jar : The jar file containing the AMI
     samples class files for Java
```

Local host and repository files

The AMI uses a *repository file* and a *local host file*. Their location and names must be specified to the AMI.

Default location

The default directory for the files on UNIX is:

```
/usr/mqm/amt         (AIX)

/opt/mqm/amt         (Solaris)
```

On Windows, the default location is a directory called `\amt` under the user specified MQSeries file directory. For example, if MQSeries is installed in the `C:\MQSeries` directory, the default directory for the AMI data files on Windows NT is:

```
C:\MQSeries\amt
```

Default names

The default name for the repository file is `amt.xml`, and the default name for the host file is `amthost.xml`.

A sample host file (which can be used as a default) is provided in the correct location. A sample repository file is located in the following directory:

```
/amt/samp            (UNIX)

\amt\samples         (Windows)
```

Overriding the default location and names

You can override where the AMI looks for the repository and local host files by using an environment variable:

```
export AMT_DATA_PATH = /directory          (UNIX)

set AMT_DATA_PATH = X:\directory           (Windows)
```

You can override the default names of the repository and local host files by using environment variables:

```
export AMT_REPOSITORY = myData.xml         (UNIX)
export AMT_HOST = myHostFile.xml

set AMT_REPOSITORY = myData.xml            (Windows)
set AMT_HOST = myHostFile.xml
```

The directories `intlFiles` and `locales`, and the `.txt` and `.cnv` files in the `locales` directory, must be located relative to the directory containing the local host file. This applies whether you are using the default directory or have overridden it as described above.

In C++ and Java there is an extra level of flexibility in setting the location and names of the repository and local host files. You can specify the directory in which they are located by means of a name in the constructor of the AmSessionFactory class:

```
AmSessionFactory(name);
```

This name is equivalent to the AMT_DATA_PATH environment variable. If set, the name of the AmSessionFactory takes precedence over the AMT_DATA_PATH environment variable.

The repository and local host file names can be set using methods of the AmSessionFactory class:

```
setRepository(name);
setLocalHost(name);
```

These AmSessionFactory methods take precedence over the AMT_REPOSITORY and AMT_HOST environment variables.

Once an AmSession has been created using an AmSessionFactory, the repository and local host file names and location are set for the complete life of that AmSession.

Local host file

An AMI installation must have a local host file. It defines the mapping from a connection name (default or repository defined) to the name of the MQSeries queue manager that you want to connect to on your local machine.

If you are not using a repository, or are opening (or initializing) a session using a policy that does not define a connection, the connection name is assumed to be `defaultConnection`. Using the sample `amthost.xml` file, as shown below, this maps to an empty string that defines a connection with the default queue manager.

```
<?xml version="1.0" encoding="UTF-8"?>
<queueManagerNames
        defaultConnection = ""
        connectionName1   = "queueManagerName1"
        connectionName2   = "queueManagerName2"
/>
```

To change the default connection to a named queue manager of your choice, such as ‘QMNAME’, edit the local host file to contain the following string:

```
defaultConnection = "QMNAME"
```

If you want a repository defined connection name, such as `connectionName1`, to provide a connection to queue manager ‘QMNAME1’, edit the local host file to contain the following string:

```
connectionName1   = "QMNAME1"
```

The repository connection names are not limited to the values shown (`connectionName1` and `connectionName2`). Any name can be used provided it is unique in both the repository and local host files, and consistent between the two.

Repository file

You can operate an AMI installation with or without a repository file. If you are using a repository file, such as the sample `amt.xml` file, you must have a corresponding `amt.dtd` file in the same directory (the local host file must be in this directory as well).

The repository file provides definitions for policies and services. If you do not use a repository file, AMI uses its built-in definitions. For more information, see Chapter 13, “Defining services and policies” on page 279.

The administration tool

Installation

The AMI administration tool is for use on Windows NT Version 4 only. It is installed and started as follows.

Using Info-ZIP’s UnZip program (or a similar program), unzip the file `ma0g.zip` into a suitable directory. The AMI administration tool files are installed in sub-directory `\amt` within that directory.

To start the AMI administration tool, double-click on the file `\amt\tool\amitool.bat` in the installation directory.

To verify that the tool has been installed correctly, click on **Open** in the **File** menu, navigate to the `\amt\tool` directory, and open the file `amiSample.xml`. You should see a number of services and policies in the navigation pane on the left. Select one of them by clicking on it, and you should see its attributes displayed in the pane on the right.

Operation

The administration tool enables you to create definitions for:

Service points	used to create sender or receiver services
Distribution lists	must include at least one sender service
Publishers	must include a sender service as the broker service
Subscribers	must include sender and receiver services as the broker and receiver services
Policies	contain sets of attributes: initialization, general, send, receive, publish, subscribe

The default attributes provided by the tool are as specified in “Service definitions” on page 282 and “Policy definitions” on page 284.

When you have entered the definitions you require, select **Save** in the **File** menu to save them as an XML-format repository file. It is recommended that you define all your services and policies in the same repository file.

The repository file must be copied to a location where it can be accessed by the AMI (see “Local host and repository files” on page 272). If the Application Messaging Interface is on the same system as the tool, the repository file can be copied to the AMI directory. Otherwise, the repository file must be transferred to that system using a method such as file sharing or FTP.

Note: In order to open an existing repository file (including the `amt.xml` file provided in the samples directory), the repository file and the `amt.dtd` file must both be in the same directory.

Further information can be found in the AMI administration tool online help.

Connecting to MQSeries

You can connect to MQSeries, the transport layer, using an MQSeries server or an MQSeries client. Using the default policy, the AMI automatically detects whether it should connect directly or as a client. If you have an installation that has both an MQSeries client and an MQSeries queue manager, and you want the AMI to use the client for its connection, you must specify the Connection Type as Client in the policy initialization attributes (see “Policy definitions” on page 284).

Using MQSeries Integrator Version 1

If you are using the AMI with MQSeries Integrator Version 1, the Service Type for the sender service point must be defined in the repository as MQSeries Integrator V1 (see “Service definitions” on page 282). This causes an MQRFH header containing application group and message type name/value elements to be added to a message when it is sent.

The Application Group definition is included in the policy send attributes (see “Policy definitions” on page 284). The message type is defined as the message format value set in the message object (using **amMsgSetFormat**, for example). If this is set to AMFMT_NONE, the message type is defined as the Default Format for the sender service point (a maximum of eight characters in MQSeries). If you wish to specify the message type directly, you must do this explicitly using the **amMsgAddElement** function in C, or the equivalent **addElement** method in C++ and Java. This allows you to add a message type that differs from the message format, and is more than eight characters long.

Using MQSeries Publish/Subscribe

If you want to use the publish/subscribe functions of the AMI, you must have MQSeries Publish/Subscribe installed (see the *MQSeries Publish/Subscribe User's Guide*). The Service Type for the sender and receiver service points used by the publisher and subscriber must be defined in the repository as MQRFH (see “Service definitions” on page 282). This causes an MQRFH header containing publish/subscribe name/value elements to be added to a message when it is sent.

Creating default MQSeries objects

The Application Messaging Interface makes use of default MQSeries objects, which must be created prior to using the AMI. This can be done by running the MQSC script `amtsdfts.tst`. (You might want to edit this file first, to suit the requirements of your installation.)

First start the local queue manager by typing the following at a command line:

```
strmqm {QMName}
```

where `{QMName}` is the name of your MQSeries queue manager.

Then run the default MQSC script by typing one of the following:

```
runmqsc {QMName} < {Location}/amtsdfts.tst    (UNIX)

runmqsc {QMName} < {Location}\amtsdfts.tst    (Windows)
```

where `{QMName}` is the name of your MQSeries queue manager and `{Location}` is the location of the `amtsdfts.tst` file.

Running the sample programs

Sample programs are provided to illustrate the use of the Application Messaging Interface.

It is recommended that you run one or more of the sample programs to verify that you have installed the Application Messaging Interface correctly.

Setting up the samples

Before you can run the sample programs, there are a number of actions to be taken.

MQSeries objects

The sample programs require some MQSeries objects to be defined. This can be done with an MQSeries MQSC file, `amtsamp.tst`, which is shipped with the samples.

First start the local queue manager by typing the following at a command line:

```
strmqm {QMName}
```

where `{QMName}` is the name of your MQSeries queue manager.

Then run the sample MQSC script by typing one of the following:

```
runmqsc {QMName} < {Location}/amtsamp.tst    (UNIX)

runmqsc {QMName} < {Location}\amtsamp.tst    (Windows)
```

where `{QMName}` is the name of your MQSeries queue manager and `{Location}` is the location of the `amtsamp.tst` file.

Repository and host files

Copy the sample repository file, `amt.xml`, into the default location for your platform (see “Local host and repository files” on page 272).

Modify the host file so that your MQSeries queue manager name, {QMName}, is known as `defaultConnection`.

MQSeries Publish/Subscribe broker

If you are running any of the publish/subscribe samples, you must also start the MQSeries Publish/Subscribe broker. Type the following at a command line:

```
strmqbrk -m {QMName}
```

where `{QMName}` is the name of your MQSeries queue manager.

The sample programs

There are six basic sample programs, performing approximately the same function in C, C++ and Java. Consult the source code to find out how the programs achieve this functionality. The C samples are provided for both the high-level interface and the object interface.

Table 3. The sample programs

Description	C high-level	C object-level	C++	Java
A sample that sends a datagram message, expecting no reply.	amtshsnd	amtsosnd	SendAndForget	SendAndForget
A sample that receives a message, with no selection.	amtshrcv	amtsorcv	Receiver	Receiver
A sample that sends a request and receives a reply to this request (a simple client program).	amtshclt	amtsoclt	Client	Client
A sample that receives requests and sends replies to these requests (a simple server program).	amtshsvr	amtsosvr	Server	Server
A sample that periodically publishes information on the weather.	amtshpub	amtsopub	Publisher	Publisher
A sample that subscribes to information on the weather, and receives publications based on this subscription.	amtshsub	amtsosub	Subscriber	Subscriber

To find the source code and the executables for the samples, see "Directory structure" on page 263 (AIX), page 267 (Solaris), and page 270 (Windows).

Setting the runtime environment

Before you run the AMI samples, make sure that you have set up the runtime environment. See “Setting the runtime environment” on page 262 (AIX), page 266 (Solaris), and page 269 (Windows).

Running the C and C++ sample programs

You can run a sample program by typing the name of its executable at a command line. For example:

```
amtsosnd
```

will run the “Send and forget” sample written using the C object interface.

Running the Java sample programs

The AMI samples for Java are in a package called:

```
com.ibm.mq.amt.samples
```

In order to invoke them you need to specify the name of the sample plus its package name. For example, to run the “Send and forget” sample use:

```
java com.ibm.mq.amt.samples.SendAndForget
```

Chapter 13. Defining services and policies

Definitions of services and policies created by a system administrator are held in a *repository*. The Application Messaging Interface provides a tool to enable the administrator to set up new services and policies, and to specify their attributes (see “The administration tool” on page 274).

This chapter contains:

- “Services and policies”
- “Service definitions” on page 282
- “Policy definitions” on page 284

Services and policies

A repository file contains definitions for *policies* and *services*. A service is the generic name for any object to which a send or receive request can be issued, that is:

- Sender
- Receiver
- Distribution list
- Publisher
- Subscriber

Sender and receiver definitions are represented in the repository by a single definition called a *service point*.

Policies, and services other than distribution lists, can be created with or without a corresponding repository definition; distribution lists can be created only with a corresponding repository definition.

To create a service or policy using the repository, the repository must contain a definition of the appropriate type with a name that matches the name specified by the application. To create a sender object named ‘DEBITS’ (using **amSesCreateSender** in C, for example) the repository must have a service point definition named ‘DEBITS’.

Policies and services created with a repository have their contents initialized from the named repository definition.

If the repository does not contain a matching name, a warning is issued (such as AMRC_POLICY_NOT_IN_REPOS). The service or policy is then created without using the repository (unless it is a distribution list).

Policies and services created without a repository (either for the above reason, or because the repository is not used), have their contents initialized from one of the system provided definitions (see “System provided definitions”).

Definition names in the repository must not start with the characters ‘AMT’ or ‘SYSTEM’.

System provided definitions

The AMI provides a set of definitions for creating services and policies without reference to a repository.

Table 4. System provided definitions

Definition	Description
AMT.SYSTEM.POLICY	This provides a policy definition with the defaults specified in “Policy definitions” on page 284, except that Wait Interval Read Only is set to ‘No’ in the Receive attributes.
AMT.SYSTEM.SYNCPOINT.POLICY	This provides a policy definition the same as AMT.SYSTEM.POLICY, except that Syncpoint is set to ‘Yes’ in the Send attributes and in the Receive attributes.
AMT.SYSTEM.SENDER	This provides a sender definition with the defaults specified in “Service definitions” on page 282, with the Queue Name the same as the Sender object.
AMT.SYSTEM.RESPONSE.SENDER	This provides a sender definition the same as AMT.SYSTEM.SENDER, except that Definition Type, Queue Name and Queue Manager Name are set to ‘Undefined’ (that is, set when used).
AMT.SYSTEM.RECEIVER	This provides a receiver definition the same as AMT.SYSTEM.SENDER.
AMT.SYSTEM.PUBLISHER	This provides a publisher definition in which the Broker Service has the same name as the Publisher object.
AMT.SYSTEM.SUBSCRIBER	This provides a subscriber definition in which the Sender Service has the same name as the Subscriber object, and the Receiver Service has the same name with the suffix ‘.RECEIVER’.

System default objects

A set of system default objects is created at session creation time. This removes the overhead of creating the objects from applications using these defaults. The system default objects are available for use from the high-level and object-level interfaces in C. They cannot be accessed using C++ or Java (these languages can use the built-in definitions to create an equivalent set of objects if required).

The default objects are created using the system provided definitions, as shown in the following table.

Table 5. System default objects

Default object	**Definition**
SYSTEM.DEFAULT.POLICY	AMT.SYSTEM.POLICY
SYSTEM.DEFAULT.SYNCPOINT.POLICY	AMT.SYSTEM.SYNCPOINT.POLICY
SYSTEM.DEFAULT.SENDER	AMT.SYSTEM.SENDER
SYSTEM.DEFAULT.RESPONSE.SENDER	AMT.SYSTEM.RESPONSE.SENDER
SYSTEM.DEFAULT.RECEIVER	AMT.SYSTEM.RECEIVER
SYSTEM.DEFAULT.PUBLISHER	AMT.SYSTEM.PUBLISHER
SYSTEM.DEFAULT.SUBSCRIBER	AMT.SYSTEM.SUBSCRIBER
SYSTEM.DEFAULT.SEND.MESSAGE	N/A
SYSTEM.DEFAULT.RECEIVE.MESSAGE	N/A

The default objects can be used explicitly using the AMI constants (see Appendix B, “Constants” on page 313), or used to provide defaults if a particular parameter is omitted (by specifying NULL, for instance).

Handles are also provided for these objects, for use from the object interface (see Appendix B, “Constants” on page 313). Note that the first parameter on a call must be a real handle; you cannot use a default handle in this case.

Service definitions

Service point (sender/receiver)

Table 6. Service point (sender/receiver)

Attribute	Comments
Name	Mandatory name, specified an AMI calls. 1
Queue Name	Name of the queue representing the service that messages are sent to or received from. Required if the Definition Type is 'Predefined'. 2
Queue Manager Name	Name of the queue manager that owns Queue Name. If blank, the local queue manager name is used. 2
Model Queue Name	Name of a model queue definition used to create a dynamic queue (normally a Reply Service to receive response messages). Required if the Definition Type is 'Dynamic'. 2
Dynamic Queue Prefix	Name of a prefix used when creating a dynamic queue from Model Queue Name. Required if the Definition Type is 'Dynamic' and the last non-blank character in positions 1 to 33 is `'*'`. The `'*'` is replaced by a string that guarantees that the name generated is unique. 2
Definition Type	Defines how the AMI obtains the queue name for the service point. If set to 'Predefined' (the default), the Queue Name and Queue Manager Name as specified above are used. If set to 'Dynamic', the Model Queue Name and Dynamic Queue Prefix are used to create a dynamic queue.
Service Type	Defines the header (if any) that is sent with the message data, and the parameters within the header. Set to 'Native' for a native MQ service (default). Set to 'MQSeries Integrator V1' for MQSeries Integrator Version 1 (adds the OPT_APP_GROUP and OPT_MSG_TYPE fields to the MQRFH header). Set to 'RF Header V1' for MQSeries Publish/Subscribe applications.
Default Format	Optional format name to insert in the MQMD, if a format is not passed by the application. Also used as the MsgType when the service is an MQSeries Integrator Version 1 broker, if AMFMT_NONE is set in the message object and the MsgType has not been added explicitly (using **amMsgAddElement** or equivalent). 3
CCSID	Coded character set identifier of the destination application. Can be used by sending applications to prepare a message in the correct CCSID for the destination. Leave blank if the CCSID is unknown (the default), or set to the CCSID number.
Encoding	Integer encoding of the destination application. Can be used by sending applications to prepare a message in the correct encoding for the destination. Set to 'Unspecified' (the default), 'Reversed', 'Normal', 'Reversed With 390 Floating Point', or 'Normal With 390 Floating Point'.

Notes:

1 The name is a maximum of 256 characters, and can contain the following characters: `A-Z, a-z, 0-9, '.', '/', '_' and '%'`.

2 The name is a maximum of 48 characters, and can contain the following characters: `A-Z, a-z, 0-9, '.', '/', '_' and '%'`.

3 The name is a maximum of 8 characters, and can contain any character from a single byte character set (it is recommended that the characters are restricted to `A-Z, 0-9`).

Distribution list

Table 7. Distribution list

Attribute	Comments
Name	Mandatory name, specified an AMI calls. **1**
Available Service Points	List of service points that make up the distribution list. They must be valid service point names.
Note: **1** The name is a maximum of 256 characters, and can contain the following characters: `A-Z`, `a-z`, `0-9`, `'.'`, `'/'`, `'_'` and `'%'`.	

Publisher

Table 8. Publisher

Attribute	Comments
Name	Mandatory name, specified an AMI calls. **1**
Broker Service	The name of a sender service that defines the publish/subscribe broker. It must be a valid service point name.
Note: **1** The name is a maximum of 256 characters, and can contain the following characters: `A-Z`, `a-z`, `0-9`, `'.'`, `'/'`, `'_'` and `'%'`.	

Subscriber

Table 9. Subscriber

Attribute	Comments
Name	Mandatory name, specified an AMI calls. **1**
Broker Service	The name of the sender service that defines the publish/subscribe broker. It must be a valid service point name.
Receiver Service	The name of the receiver service that defines where publication messages are sent. It must be a valid service point name.
Note: **1** The name is a maximum of 256 characters, and can contain the following characters: `A-Z`, `a-z`, `0-9`, `'.'`, `'/'`, `'_'` and `'%'`.	

Policy definitions

Initialization attributes

Table 10. Initialization attributes

Attribute	Comments
Name	Mandatory policy name, specified an AMI calls. **1**
Connection Name	Name of the logical connection used to generate the queue manager to which connection is made. If it is omitted, the local default queue manager is used. **2**
Connection Mode	If Connection Mode is set to 'Real' (the default), Connection Name is used as the queue manager name for connection. If Connection Mode is set to 'Logical', Connection Name is used as a key to the host file on the system where the application is running that maps Connection Name to a queue manager name. This allows the same application running on different systems in the network to connect to different local queue managers.
Connection Type	Used only if Connection Mode is set to 'Real'. If Connection Type is set to 'Auto' (the default), the application automatically detects if it should connect directly, or as a client. If Connection Type is 'Client', the application connects as a client. If Connection Type is 'Server', the application connects directly to the queue manager.
Trusted Option	If set to 'Normal' (the default), no fastpath is used. If set to 'Trusted', the application can use fastpath facilities that might compromise integrity.

Notes:

1 The name is a maximum of 256 characters, and can contain the following characters: `A-Z, a-z, 0-9, '.', '/', '_' and '%'.`

2 The name is a maximum of 48 characters, and can contain the following characters: `A-Z, a-z, 0-9, '.', '/', '_' and '%'.`

General attributes

Table 11. General attributes

Attribute	Comments
Message Context	Defines how the message context is set in messages sent by the application. The default is 'Set By Queue Manager' (the queue manager sets the context). If set to 'Pass Identity', the identity of the request message is passed to any output messages. If set to 'Pass All', all the context of the request message is passed to any output messages. If set to 'No Context', no context is passed.
Syncpoint	The send or receive is part of a unit of work (default is 'No').

Send attributes

Table 12. Send attributes

Attribute	Values	Default	Comments
Implicit Open	Yes No	Yes	The queue is opened implicitly (must be set to 'Yes' for the C high-level interface). [1]
Leave Queue Open	Yes No	Yes	The queue is left open after use. [1]
Priority	0-9 As Transport	As Transport	The priority set in the message (the default uses the value from the queue definition).
Persistence	Yes No As Transport	As Transport	The persistence set in the message (the default uses the value from the queue definition).
Expiry Interval	0-999999999 Unlimited	Unlimited	A period of time (in tenths of a second) after which the message will not be delivered.
Retry Count	0-999999999	0	The number of times a send will be retried if the return code gives a temporary error. Retry will be attempted under the following conditions: Queue full, Queue disabled for put, Queue in use.
Retry Interval	0-999999999	1000	The interval (in milliseconds) between each retry.
Response Correl Id	Message Id Correl Id	Message Id	Response or report messages have their Correl Id set to the Message Id or Correl Id of the request message.
Exception Action	Discard DLQ	DLQ	If a message cannot be delivered it will be discarded or put to the dead-letter queue.
Report Data	Report With Data With Full Data	Report	Specifies if data (first 100 bytes) or full data is included in a report messages. Default is 'Report' (no data).
Report Type Exception	Yes No	No	Specifies if Exception reports are required.
Report Type COA	Yes No	No	Specifies if Confirm on Arrival reports are required.
Report Type COD	Yes No	No	Specifies if Confirm on Delivery reports are required.
Report Type Expiry	Yes No	No	Specifies if Expiry reports are required.
Segmentation	Yes No	No	Segmentation of the message is allowed.
Application Group	Name		Optional application group name when the service represents an MQSeries Integrator Version 1 broker. [2]

Notes:

[1] If Implicit Open is 'Yes' and Leave Open is 'No', MQPUT1 is used for send operations.

[2] The name is a maximum of 48 characters, and can contain the following characters: `A-Z`, `a-z`, `0-9`, `'.'`, `'/'`, `'_'` and `'%'`.

Receive attributes

Table 13. Receive attributes

Attribute	Values	Default	Comments
Implicit Open	Yes No	Yes	The queue is opened implicitly (must be set to 'Yes' for the C high-level interface).
Leave Queue Open	Yes No	Yes	The queue is left open after use.
Delete On Close	Yes No Purge	Yes	Dynamic queues are deleted when closed. 'Purge' causes deletion even if there are messages on the queue.
Wait Interval	0-999999999 Unlimited,	Unlimited	A period of time (in milliseconds) that the receive waits for a message to be available.
Wait Interval Read Only	Yes No	Yes	If set to 'No', an application can override the Wait Interval value in the policy object.
Convert	Yes No	Yes	The message is code page converted by the message transport.
Wait For Whole Group	Yes No	Yes	All messages in a group must be available before any message is returned by the receive.
Handle Poison Message	Yes No	Yes	Enables poison message handling. 1
Accept Truncated Message	Yes No	Yes	Truncated messages are accepted.
Open Shared	Yes No	Yes	The queue is opened as a shared queue.

Note:

1 A poison message is one for which the count of the number of times it has been backed-out exceeds the maximum backout-limit specified by the underlying MQSeries transport queue object. If poison message handling is enabled during a receive request the AMI will handle it as follows:

If a poison message is successfully requeued to the backout-requeue queue (specified by the underlying MQSeries transport queue), the message is returned to the application with completion code MQCC_WARNING and reason code MQRC_BACKOUT_LIMIT_ERR.

If a poison message requeue attempt (as described above) is unsuccessful, the message is returned to the application with completion code MQCC_WARNING and reason code MQRC_BACKOUT_REQUEUE_ERR.

If a poison message is part of a message group (and not the only message in the group), no attempt is made to requeue the message. The message is returned to the application with completion code MQCC_WARNING and reason code MQRC_GROUP_BACKOUT_LIMIT_ERR.

Publish attributes

Table 14. Publish attributes

Option	Values	Default	Comments
Retain	Yes No	No	The publication is retained by the broker.
Publish To Others Only	Yes No	No	The publication is not sent to the publisher if it has subscribed to the same topic (used for conference-type applications).
Suppress Registration	Yes No	No	Implicit registration of the publisher is suppressed.
Publish Locally	Yes No	No	The publication is sent to subscribers at the local broker only.
Accept Direct Requests	Yes No	No	The publisher supports direct requests from subscribers.
Anonymous Registration	Yes No	No	The publisher registers anonymously.
Use Correl Id As Id	Yes No	No	The Correl Id is used by the broker as part of the publisher’s identity.

Subscribe attributes

Table 15. Subscribe attributes

Option	Values	Default	Comments
Subscribe Locally	Yes No	No	The subscriber is sent publications that were published with the Publish Locally option, at the local broker only.
New Publications Only	Yes No	No	The subscriber is not sent existing retained publications when it registers.
Publish On Request Only	Yes No	No	The subscriber is not sent retained publications unless it requests them by using Request Update.
Inform If Retained	Yes No	Yes	The broker informs the subscriber if a publication is retained.
Unsubscribe All	Yes No	No	All topics for this subscriber are to be deregistered.
Anonymous Registration	Yes No	No	The subscriber registers anonymously.
Use Correl Id As Id	Yes No	No	The Correl Id is used by the broker as part of the subscriber’s identity.

Chapter 14. Problem determination

This chapter shows you how to use the trace facility in the Application Messaging Interface, and gives some information about finding the causes of problems. See:

- "Using trace"
- "When your AMI program fails" on page 297

Using trace

The Application Messaging Interface includes a trace facility to help identify what is happening when you have a problem. It shows the paths taken when you run your AMI program. Unless you have a problem, you are recommended to run with tracing set off to avoid any unnecessary overheads on your system resources.

There are three environment variables that you set to control trace:

AMT_TRACE
AMT_TRACE_PATH
AMT_TRACE_LEVEL

You set these variables in one of two ways.

1. From a command prompt. It is effective locally, so you must then start your AMI program from this prompt.
2. By putting the information into your system startup file; this is effective globally. To do this:
   - Select Main -> Control Panel on Windows NT and Windows 98
   - Edit your `.profile` file on UNIX systems

When deciding where you want the trace files written, ensure that the user has sufficient authority to write to, not just read from, the disk.

If you have tracing switched on, it will slow down the running of your AMI program, but it will not affect the performance of your MQSeries environment. When you no longer need a trace file, it is your responsibility to delete it. You must stop your AMI program running to change the status of the AMT_TRACE variable. The AMI trace environment variable is different to the trace environment variable used within the MQSeries range of products. Within the AMI, the trace environment variable turns tracing on. If you set the variable to a string of characters (any string of characters) tracing will remain switched on. It is not until you set the variable to NULL that tracing is turned off.

Trace filename and directory

The trace file name takes the form AMTnnnnn.trc, where nnnnn is the ID of the AMI process running at the time.

Commands on UNIX

export AMT_TRACE_PATH=/directory
: Sets the trace directory where the trace file will be written.

unset AMT_TRACE_PATH
: Removes the AMT_TRACE_PATH environment variable; the trace file is written to the current working directory (when the AMI program was started).

echo $AMT_TRACE_PATH
: Displays the current setting of the trace directory path.

export AMT_TRACE_LEVEL=n
: Sets the trace level, where n is an integer from 0 through 9. 0 represents minimal tracing, and 9 represents a fully detailed trace.

  In addition, you can suffix the value with a + (plus) or - (minus) sign. Using the plus sign, the trace includes all control block dump information and all informational messages. Using the minus sign includes only the entry and exit points in the trace with no control block information or text output to the trace file.

unset AMT_TRACE_LEVEL
: Removes the AMT_TRACE_LEVEL environment variable. The trace level is set to its default value of 2.

echo $AMT_TRACE_LEVEL
: Displays the current setting of the trace level.

export AMT_TRACE=xxxxxxxx
: This sets tracing ON. You switch tracing on by putting one or more characters after the '=' sign. For example:

```
export AMT_TRACE=yes
export AMT_TRACE=no
```

  In both of these examples, tracing will be set ON.

unset AMT_TRACE
: Sets tracing off

echo $AMT_TRACE
: Displays the contents of the environment variable.

Commands on Windows:

SET AMT_TRACE_PATH=drive:\directory
: Sets the trace directory where the trace file will be written.

SET AMT_TRACE_PATH=
: Removes the AMT_TRACE_PATH environment variable; the trace file is written to the current working directory (when the AMI program was started).

SET AMT_TRACE_PATH
: Displays the current setting of the trace directory.

SET AMT_TRACE_LEVEL=n
: Sets the trace level, where n is an integer from 0 through 9. 0 represents minimal tracing, and 9 represents a fully detailed trace.

  In addition, you can suffix the value with a + (plus) or - (minus) sign. Using the plus sign, the trace includes all control block dump information and all informational messages. Using the minus sign includes only the entry and exit points in the trace with no control block information or text output to the trace file.

SET AMT_TRACE_LEVEL=
: Removes the AMT_TRACE_LEVEL environment variable. The trace level is set to its default value of 2.

SET AMT_TRACE_LEVEL
: Displays the current setting of the trace level.

SET AMT_TRACE=xxxxxxxx
: This sets tracing ON. You switch tracing on by putting one or more characters after the '=' sign. For example:

```
SET AMT_TRACE=yes
SET AMT_TRACE=no
```

In both of these examples, tracing will be set ON.

SET AMT_TRACE=
: Sets tracing OFF

SET AMT_TRACE
: Displays the contents of the environment variable.

C++ and Java

For these language bindings there is more control over the production of trace. In each case, the AmSessionFactory has two methods which control trace:

1. setTraceLocation(location);
2. setTraceLevel(level);

The behavior of these methods matches exactly the behavior of the environment variables:

1. AMT_TRACE_PATH
2. AMT_TRACE_LEVEL

Once an AmSession has been created using an AmSessionFactory, the trace level and location are set for the complete life of that AmSession.

If set, the values of the properties in the AmSessionFactory take precedence over any AMT trace environment variables.

Example trace

The example trace below shows 'typical' trace output.

```
Trace for program d:\output\bin\amITSR.exe <<< AMT trace >>>
 started at Sat Jun 12 08:28:33 1999

@(!) <<< *** Code Level is 1.0.0 *** >>>
   !(03787) BuildDate Jun 11 1999
   !(03787) Trace Level is 2

(03787)@08:28:33.728
   -->xmq_xxxInitialize

   ---->ObtainSystemCP
   !(03787) Code page is 437

   <----ObtainSystemCP (rc = 0)

   <--xmq_xxxInitialize (rc = 0)

   -->amSessCreateX

   ---->amCheckAllBlanks()

   <----amCheckAllBlanks() (rc = 0)

   ---->amCheckValidName()

   <----amCheckValidName() (rc = 1)
   !(03787) Session name is: plenty

   ---->amHashTableCreate()

   <----amHashTableCreate() (rc = AM_ERR_OK)

   ---->amSessClearErrorCodes

   <----amSessClearErrorCodes (rc = 0)

  ...

   ---->amMaSrvCreate
   !(03787) Service object created [9282320]

   <----amMaSrvCreate (rc = AM_ERR_OK)

   ---->amMaSrvSetSessionHandle
   !(03787) Object handle[9282320]

   <----amMaSrvSetSessionHandle (rc = AM_ERR_OK)

   ---->amHashTableAddHandle()

   <----amHashTableAddHandle() (rc = AM_ERR_OK)
   ---->amMaSrvCreate
   !(03787) Service object created [9285144]

   <----amMaSrvCreate (rc = AM_ERR_OK)

   ---->amMaSrvSetSessionHandle
   !(03787) Object handle[9285144]

   <----amMaSrvSetSessionHandle (rc = AM_ERR_OK)

   ---->amHashTableAddHandle()

   <----amHashTableAddHandle() (rc = AM_ERR_OK)

(03787)@08:28:33.738
   ---->amMaSrvCreate
   !(03787) Service object created [9287968]

   <----amMaSrvCreate (rc = AM_ERR_OK)

   ---->amMaSrvSetSessionHandle
   !(03787) Object handle[9287968]

   <----amMaSrvSetSessionHandle (rc = AM_ERR_OK)

   ---->amHashTableAddHandle()

   <----amHashTableAddHandle() (rc = AM_ERR_OK)

   ---->amMaSrvCreate
   !(03787) Service object created [9290792]

   <----amMaSrvCreate (rc = AM_ERR_OK)

   ---->amMaSrvSetSessionHandle
   !(03787) Object handle[9290792]

   <----amMaSrvSetSessionHandle (rc = AM_ERR_OK)

   ---->amHashTableAddHandle()

   <----amHashTableAddHandle() (rc = AM_ERR_OK)

   ---->amMaSrvCreate

   !(03787) Service object created [9293616]

   <----amMaSrvCreate (rc = AM_ERR_OK)

   ---->amMaSrvSetSessionHandle
   !(03787) Object handle[9293616]

   <----amMaSrvSetSessionHandle (rc = AM_ERR_OK)

   ---->amHashTableAddHandle()

   <----amHashTableAddHandle() (rc = AM_ERR_OK)
```

```
---->amMaSrvCreate
!(03787) Service object created [9296440]

<----amMaSrvCreate (rc = AM_ERR_OK)

---->amMaSrvSetSessionHandle
!(03787) Object handle[9296440]

<----amMaSrvSetSessionHandle (rc = AM_ERR_OK)

---->amMaSrvSetSubReceiverHandle
!(03787) Object handle[9293616]

<----amMaSrvSetSubReceiverHandle (rc = AM_ERR_OK)

---->amMaMsgCreate
!(03787) message object created -[10420288]

<----amMaMsgCreate (rc = AM_ERR_OK)

---->amHashTableAddHandle()

<----amHashTableAddHandle() (rc = AM_ERR_OK)

---->amMaMsgCreate
!(03787) message object created -[10432440]

<----amMaMsgCreate (rc = AM_ERR_OK)

---->amHashTableAddHandle()

<----amHashTableAddHandle() (rc = AM_ERR_OK)

---->amMaPolCreate
!(03787) policy object created.
!(03787) policy object initialized.

<----amMaPolCreate (rc = AM_ERR_OK)

---->amHashTableAddHandle()

<----amHashTableAddHandle() (rc = AM_ERR_OK)

---->amMaPolCreate
!(03787) policy object created.
!(03787) policy object initialized.

<----amMaPolCreate (rc = AM_ERR_OK)

---->amHashTableAddHandle()

<----amHashTableAddHandle() (rc = AM_ERR_OK)
```

```
   ---->amMaPolSetIntProps
   !(03787) Object handle[10446656]
   !(03787) [AMPOL_IPR_APR_CON_CNT] set to [0x1]

(03787)@08:28:33.748
   <----amMaPolSetIntProps (rc = AM_ERR_OK)

   ---->amMaPolSetStringProp
   !(03787) Object handle[10446656]
   !(03787) [AMPOL_SPR_APR_MGR_NAME] set to [plenty]

   <----amMaPolSetStringProp (rc = AM_ERR_OK)

   ---->amMaPolSetStringProp
   !(03787) Object handle[10446656]
   !(03787) [AMPOL_SPR_APR_CON_NAME] set to [plenty]

   <----amMaPolSetStringProp (rc = AM_ERR_OK)

   ---->amMaSrvSetStringProp
   !(03787) Object handle[9282320]
   !(03787) [AMSRV_SPR_QUEUE_NAME] set to [SYSTEM.DEFAULT.SENDER]

   <----amMaSrvSetStringProp (rc = AM_ERR_OK)

   ---->amMaSrvSetStringProp
   !(03787) Object handle[9285144]
   !(03787) [AMSRV_SPR_QUEUE_NAME] set to []

   <----amMaSrvSetStringProp (rc = AM_ERR_OK)

   ---->amMaSrvSetStringProp
   !(03787) Object handle[9287968]
   !(03787) [AMSRV_SPR_QUEUE_NAME] set to [SYSTEM.DEFAULT.RECEIVER]

   <----amMaSrvSetStringProp (rc = AM_ERR_OK)

   ---->amMaSrvSetStringProp
   !(03787) Object handle[9290792]
   !(03787) [AMSRV_SPR_QUEUE_NAME] set to [SYSTEM.DEFAULT.PUBLISHER]

   <----amMaSrvSetStringProp (rc = AM_ERR_OK)

   ---->amMaSrvSetStringProp
   !(03787) Object handle[9293616]
   !(03787) [AMSRV_SPR_QUEUE_NAME] set to [SYSTEM.DEFAULT.SUBSCRIBER]

   <----amMaSrvSetStringProp (rc = AM_ERR_OK)

   ---->amMaPolSetIntProps
   !(03787) Object handle[10451304]
   !(03787) [AMPOL_IPR_SMO_SYNCPOINT] set to [0xc030003]

   <----amMaPolSetIntProps (rc = AM_ERR_OK)
```

```
    ---->amMaPolSetIntProps
    !(03787) Object handle[10451304]
    !(03787) [AMPOL_IPR_RMO_SYNCPOINT] set to [0xd060002]

    <----amMaPolSetIntProps (rc = AM_ERR_OK)

    ---->amActivateFiles
    !(03787) No DATAPATH specified from API
    !(03787) No repository FILE specified from API
    !(03787) Repository[H:\MQSeries\amt\\amt.xml]
    !(03787) Repository ACTIVE
    !(03787) No local host FILE specified from API
    !(03787) Local Host[H:\MQSeries\amt\\amthost.xml]
    !(03787) Local Host File ACTIVE

    <----amActivateFiles (rc = 1)

    ---->amErrTranslate

    <----amErrTranslate (rc = 0)

    <--amSessCreateX (rc = 0)

...
```

When your AMI program fails

Reason Codes

When an AMI function call fails, it reports the level of the failure in the completion code of the call. AMI has three completion codes:

AMCC_OK	The call completed successfully
AMCC_WARNING	The call completed with unexpected results
AMCC_FAILED	An error occurred during processing

In the last two cases, AMI supplies a reason code that provides an explanation of the failure. A list of AMI reason codes is given in Appendix A, “Reason codes” on page 301.

In addition, if MQSeries is the reason for the failure, AMI supplies a secondary reason code. The secondary reason codes can be found in the *MQSeries Application Programming Reference* book.

First failure symptom report

A *first failure symptom* report is produced for unexpected and internal errors. This report is found in a file named `AMTnnnnn.FDC`, where `nnnnn` is the ID of the AMI process that is running at the time. You find this file in the working directory from which you started your AMI program, or the name of the path specified in the AMT_TRACE_PATH environment variable. If you receive a first failure symptom report you should contact IBM support personnel.

Other sources of information

AMI makes use of MQSeries as a transport mechanism and so MQSeries error logs and trace information can provide useful information. See the *MQSeries System Administration* manual for details of how to activate these problem determination aids.

Common causes of problems

- With the C object interface, most functions require a handle to the object they refer to. If this handle is not valid, the results are unpredictable.
- Completion code 2 (AMRC_ERROR) together with reason code 110 (AMRC_TRANSPORT_NOT_AVAILABLE) returned by **amInitialize** or **amSesOpen** (or the equivalent C++ and Java methods) normally indicates that the underlying MQSeries queue manager the AMI is attempting to use is not started (or does not exist). This might be because of a missing or incorrect xml repository file or because the data in the local host file is incorrect.
- Completion code 2 (AMRC_ERROR) together with reason code 47 (AMRC_TRANSPORT_ERR) indicates that an error was detected by the underlying MQSeries transport. The secondary reason code returned by the appropriate ‘get last error’ function for the object concerned will provide the related the MQSeries reason code. This error occurs most frequently during an attempt to open an underlying MQSeries queue object that does not exist (or has an incorrect type). This can be because it has never been created or

because a missing or incorrect xml repository file is providing an incorrect queue name.

Part 6. Appendixes

Appendix A. Reason codes

This chapter contains a description of the AMRC_* reason codes, divided into three sections according to the value of the corresponding completion code. Within each section they are in alphabetic order. For a list of reason codes in numeric order, see Appendix B, "Constants" on page 313.

In some circumstances the AMI returns a secondary reason code that comes from MQSeries, the underlying transport layer. Please refer to the *MQSeries Application Programming Reference* manual for details of these reason codes.

Reason code: OK

The following reason code is returned with completion code: AMCC_OK

AMRC_NONE
The request was successful with no error or warning returned.

Reason code: Warning

The following reason codes are returned with completion code: AMCC_WARNING

AMRC_BACKED_OUT
The unit of work has been backed out.

AMRC_BACKOUT_LIMIT_ERR
The backout count of a received message was found to have exceeded its backout limit. The message was returned to the application and was requeued to the backout requeue queue.

AMRC_BACKOUT_REQUEUE_ERR
The backout count of a received message was found to have exceeded its backout limit. The message was returned to the application. It could not be requeued to the backout requeue queue.

AMRC_CLOSE_SESSION_ERR
An error occurred while closing the session. The session is closed.

AMRC_ENCODING_INCOMPLETE
The message contains mixed values for integer, decimal, and floating point encodings, one or more of which are undefined. The encoding value returned to the application reflects only the encoding values that were defined.

AMRC_ENCODING_MIXED
The message contains mixed values for integer, decimal and floating point encodings, one or more of which conflict. An encoding value of undefined was returned to the application.

AMRC_GROUP_BACKOUT_LIMIT_ERR
The backout count of a received message was found to have exceeded its backout limit. The message was returned to the application. It was not

requeued to the backout requeue queue because it represented a single message within a group of more than one.

AMRC_MULTIPLE_REASONS
A distribution list open or send was only partially successful and returned multiple different reason codes in its underlying sender services.

AMRC_MSG_TRUNCATED
The received message that was returned to the application has been truncated.

AMRC_NO_REPLY_TO_INFO
A response sender service specified when attempting to receive a request message was not updated with reply-to information because the request message contained no reply-to information. An attempt to send a reply message using the response sender will fail.

AMRC_NOT_CONVERTED
Data conversion of the received message was unsuccessful. The message was removed from the underlying message transport layer with the message data unconverted.

AMRC_POLICY_NOT_IN_REPOS
The definition name that was specified when creating a policy was not found in the repository. The policy was created using default values.

AMRC_PUBLISHER_NOT_IN_REPOS
The definition name that was specified when creating a publisher was not found in the specified repository. The publisher was created using default values.

AMRC_RECEIVER_NOT_IN_REPOS
The definition name that was specified when creating a receiver was not found in the repository. The receiver was created using default values.

AMRC_REPOS_WARNING
A warning associated with the underlying repository data was reported.

AMRC_SENDER_NOT_IN_REPOS
The definition name that was specified when creating a sender was not found in the repository. The sender was created using default values.

AMRC_SUBSCRIBER_NOT_IN_REPOS
The definition name that was specified when creating a subscriber was not found in the repository. The subscriber was created using default values.

AMRC_TRANSPORT_WARNING
A warning was reported by the underlying (MQSeries) message transport layer. The message transport reason code can be obtained by the secondary reason code value returned from a 'GetLastError' request for the AMI object concerned.

AMRC_UNEXPECTED_RECEIVE_ERR
An unexpected error occurred after a received message was removed from the underlying transport layer. The message was returned to the application.

AMRC_UNEXPECTED_SEND_ERR
An unexpected error occurred after a message was successfully sent. Output information updated as a result of the send request should never occur.

Reason code: Failed

The following reason codes are returned with completion code: AMCC_FAILED

AMRC_BEGIN_INVALID
The begin request was not valid because there were no participating resource managers registered.

AMRC_BROWSE_OPTIONS_ERR
The specified browse options value was not valid or contained an invalid combination of options.

AMRC_CCSID_ERR
The specified coded character value was not valid.

AMRC_CCSID_PTR_ERR
The specified coded character set id pointer was not valid.

AMRC_COMMAND_ALREADY_EXISTS
A publish, subscribe, or unsubscribe command could not be added to the message because the message already contained a command element.

AMRC_CONN_NAME_NOT_FOUND
The connection name obtained from the repository was not found in the local host file.

AMRC_CORREL_ID_BUFF_LEN_ERR
The specified correlation id buffer length value was not valid.

AMRC_CORREL_ID_BUFF_PTR_ERR
The specified correlation id buffer pointer was not valid.

AMRC_CORREL_ID_LEN_ERR
The specified correlation id length value was too long.

AMRC_CORREL_ID_LEN_PTR_ERR
The specified correlation id length pointer was not valid.

AMRC_CORREL_ID_PTR_ERR
The specified correlation id pointer was not valid.

AMRC_DATA_BUFF_LEN_ERR
The specified data buffer length value was not valid.

AMRC_DATA_BUFF_PTR_ERR
The specified data buffer pointer was not valid.

AMRC_DATA_LEN_ERR
The specified data length was not valid.

AMRC_DATA_LEN_PTR_ERR
The specified data length pointer was not valid.

AMRC_DATA_OFFSET_PTR_ERR
The specified data offset pointer was not valid.

AMRC_DATA_PTR_ERR
The specified data pointer was not valid.

AMRC_DATA_SOURCE_NOT_UNIQUE
Message data for a send operation was passed in an application data buffer and was also found in the specified message object. Data can to be sent can be included in either an application buffer or a message object but not both. The message requires a reset first, to remove existing data.

AMRC_DEFN_TYPE_ERR
The definition type defined for the service point in the repository was inconsistent with the definition type of the underlying message transport queue object when it was opened.

AMRC_DEFN_TYPE_PTR_ERR
The specified definition type pointer was not valid.

AMRC_DIST_LIST_INDEX_ERR
The specified distribution list index value was not valid.

AMRC_DIST_LIST_NOT_IN_REPOS
The definition name specified for creating a distribution list was not found in the repository. The object was not created.

AMRC_DIST_LIST_NOT_UNIQUE
The specified name could not be resolved to a unique distribution list because more than one distribution list with that name exists.

AMRC_ELEM_COUNT_PTR_ERR
The specified element count pointer was not valid.

AMRC_ELEM_INDEX_ERR
The specified element index value was not valid.

AMRC_ELEM_NAME_LEN_ERR
The specified element name length value was not valid.

AMRC_ELEM_NAME_PTR_ERR
The specified element name pointer was not valid.

AMRC_ELEM_NOT_FOUND
The specified element was not found.

AMRC_ELEM_PTR_ERR
The specified element pointer was not valid.

AMRC_ELEM_STRUC_ERR
The specified element structure was not valid. The structure id, version, or a reserved field contained an invalid value.

AMRC_ELEM_STRUC_NAME_BUFF_ERR
At least one of the name buffer (length and pointer) fields in the specified element structure was not valid.

AMRC_ELEM_STRUC_NAME_ERR
At least one of the name (length and pointer) fields in the specified element structure was not valid. Ensure that the name length, pointer, and name string are valid.

AMRC_ELEM_STRUC_VALUE_BUFF_ERR
At least one of the value buffer (length and pointer) fields in the specified structure was not valid.

AMRC_ELEM_STRUC_VALUE_ERR
At least one of the value (length and pointer) fields in the specified element structure was not valid. Ensure that the value length, pointer, and value string are valid.

AMRC_ENCODING_ERR
The specified encoding value was not valid.

AMRC_ENCODING_PTR_ERR
The specified encoding pointer was not valid.

AMRC_FORMAT_BUFF_LEN_ERR
The specified format buffer length value was not valid.

AMRC_FORMAT_BUFF_PTR_ERR
The specified format buffer pointer was not valid.

AMRC_FORMAT_LEN_ERR
The specified message format string was too long.

AMRC_FORMAT_LEN_PTR_ERR
The specified format length pointer was not valid.

AMRC_FORMAT_PTR_ERR
The specified format pointer was not valid.

AMRC_GROUP_STATUS_ERR
The specified group status value was not valid.

AMRC_GROUP_STATUS_PTR_ERR
The specified group status pointer was not valid.

AMRC_HEADER_INVALID
The RFH header structure of the message was not valid.

AMRC_HEADER_TRUNCATED
The RFH header of the message was truncated.

AMRC_HOST_FILE_ERR
The contents of the local host file are not valid.

AMRC_HOST_FILENAME_ERR
The local host file name was not valid. The value of the appropriate environment variable should be corrected.

AMRC_HOST_FILE_NOT_FOUND
A local host file with the specified name was not found.

AMRC_INCOMPLETE_GROUP
The specified request failed because an attempt was made to send a message that was not in a group when the existing message group was incomplete.

AMRC_INSUFFICIENT_MEMORY
There was not enough memory available to complete the requested operation.

AMRC_INVALID_DIST_LIST_NAME
The specified distribution list name was too long, contained invalid characters, or used the reserved prefix ‘SYSTEM.’.

AMRC_INVALID_IF SERVICE_OPEN
The receiver queue name could not be set because the receiver or subscriber service was open.

AMRC_INVALID_MSG_NAME
The specified message name was too long, contained invalid characters, or used the reserved prefix ‘SYSTEM.’.

AMRC_INVALID_POLICY_NAME
The specified policy name was too long, contained invalid characters, or used the reserved prefix ‘SYSTEM.’.

AMRC_INVALID_PUBLISHER_NAME
The specified publisher service name was too long, contained invalid characters, or used the reserved prefix ‘SYSTEM.’.

AMRC_INVALID_Q_NAME
The specified queue name was too long, or contained invalid characters.

AMRC_INVALID_RECEIVER_NAME
The specified receiver service name was too long, contained invalid characters, or used the reserved prefix ‘SYSTEM.’.

AMRC_INVALID_SENDER_NAME
The specified sender service name was too long, contained invalid characters, or used the reserved prefix ‘SYSTEM.’.

AMRC_INVALID_SESSION_NAME
The specified session name was too long, contained invalid characters, or used the reserved prefix ‘SYSTEM.’.

AMRC_INVALID_SUBSCRIBER_NAME
The specified subscriber service name was too long, contained invalid characters, or used the reserved prefix ‘SYSTEM.’.

AMRC_INVALID_TRACE_LEVEL
A specified trace level was not valid.

AMRC_JAVA_CLASS_ERR
A class referenced in AMI Java code cannot be found in the AMI Java native library. This is probably due to an incompatibility between the AMI class files and the AMI Java library. (Not applicable to the C and C++ programming languages).

AMRC_JAVA_CREATE_ERR
An unexpected error occurred when creating an AMI Java object. This is probably due to an incompatibility between the AMI class files and the AMI Java library. (Not applicable to the C and C++ programming languages).

AMRC_JAVA_FIELD_ERR
A field referenced in AMI Java code cannot be found in the AMI Java native library. This is probably due to an incompatibility between the AMI class files and the AMI Java library. (Not applicable to the C and C++ programming languages).

AMRC_JAVA_JNI_ERR
An unexpected error occurred when calling the AMI Java native library. This is probably due to an incompatibility between the AMI class files and the AMI Java library. (Not applicable to the C and C++ programming languages).

AMRC_JAVA_METHOD_ERR
A method referenced in AMI Java code cannot be found in the AMI Java native library. This is probably due to an incompatibility between the AMI class files and the AMI Java library. (Not applicable to the C and C++ programming languages).

AMRC_JAVA_NULL_PARM_ERR
The AMI Java code detected a null parameter that is not valid. (Not applicable to the C and C++ programming languages).

AMRC_MSG_HANDLE_ERR
The specified message handle was not valid.

AMRC_MSG_ID_BUFF_LEN_ERR
The specified message id buffer length value was not valid.

AMRC_MSG_ID_BUFF_PTR_ERR
The specified message id buffer pointer was not valid.

AMRC_MSG_ID_LEN_ERR
The specified message id length value was not valid.

AMRC_MSG_ID_LEN_PTR_ERR
The specified message id length pointer was not valid.

AMRC_MSG_ID_PTR_ERR
The specified message id pointer was not valid.

AMRC_MSG_NOT_FOUND
The specified message was not found, so the request was not carried out.

AMRC_MSG_NOT_UNIQUE
The specified name could not be resolved to a unique message because more than one message object with that name exists.

AMRC_NAME_BUFF_LEN_ERR
The specified name buffer length value was not valid.

AMRC_NAME_BUFF_PTR_ERR
The specified name buffer pointer was not valid.

AMRC_NAME_LEN_PTR_ERR
The specified name length pointer was not valid.

AMRC_NO_MSG_AVAILABLE
No message was available for a receive request after the specified wait time.

AMRC_NO_RESP_SERVICE
The publish request was not successful because a response receiver service is required for registration and was not specified.

AMRC_NOT_AUTHORIZED
The user is not authorized by the underlying transport layer to perform the specified request.

AMRC_POLICY_HANDLE_ERR
The specified policy handle was not valid.

AMRC_POLICY_NOT_FOUND
The specified policy was not found, so the request was not carried out.

AMRC_POLICY_NOT_UNIQUE
The specified name could not be resolved to a unique policy because more than one policy with that name exists.

AMRC_PUBLISHER_NOT_UNIQUE
The specified name could not be resolved to a unique publisher because more than one publisher object with that name exists.

AMRC_Q_NAME_BUFF_LEN_ERR
The specified queue name buffer length value was not valid.

AMRC_Q_NAME_BUFF_PTR_ERR
The specified queue name buffer pointer was not valid.

AMRC_Q_NAME_LEN_ERR
The specified queue name length value was not valid.

AMRC_Q_NAME_LEN_PTR_ERR
The specified queue name length pointer was not valid.

AMRC_Q_NAME_PTR_ERR
The specified queue name pointer was not valid.

AMRC_READ_OFFSET_ERR
The current data offset used for reading bytes from a message is not valid.

AMRC_RECEIVE_BUFF_LEN_ERR
The buffer length specified for receiving data was not valid.

AMRC_RECEIVE_BUFF_PTR_ERR
The buffer pointer specified for receiving data was not valid.

AMRC_RECEIVE_DISABLED
The specified request could not be performed because the service in the underlying transport layer is not enabled for receive requests.

AMRC_RECEIVER_NOT_UNIQUE
The specified name could not be resolved to a unique receiver because more than one receiver object with that name exists.

AMRC_REPOS_ERR
An error was returned when initializing or accessing the repository. This can occur for any of the following reasons:

- The repository XML file (for instance, `amt.xml`) contains data that is not valid.
- The DTD file (`amt.dtd`) was not found or contains data that is not valid.
- The files needed to initialize the repository (located in directories `intlFiles` and `locales`) could not be located.

Check that the DTD and XML files are valid and correctly located, and that the path settings for the local host and repository files are correct.

AMRC_REPOS_FILENAME_ERR
The repository file name was not valid. The value of the appropriate environment variable should be corrected.

AMRC_REPOS_NOT_FOUND
The repository file was not found. The value of the appropriate environment variable should be corrected.

AMRC_RESERVED_NAME_IN_REPOS
The name specified for creating an object was found in the repository and is a reserved name that is not valid in a repository. The specified object was not created.

AMRC_RESP_RECEIVER_HANDLE_ERR
The response receiver service handle specified when sending a request message was not valid.

AMRC_RESP_SENDER_HANDLE_ERR
The response sender service handle specified when receiving a request message was not valid.

AMRC_RFH_ALREADY_EXISTS
A publish, subscribe, or unsubscribe command could not be added to the message because the message already contained an RFH header. The message requires a reset first, to remove existing data.

AMRC_SEND_DATA_PTR_ERR
The buffer pointer specified for sending data was not valid.

AMRC_SEND_DATA_LEN_ERR
The data length specified for sending data was not valid.

AMRC_SEND_DISABLED
The specified request could not be performed because the service in the underlying transport layer is not enabled for send requests.

AMRC_SENDER_COUNT_PTR_ERR
The specified distribution list sender count pointer was not valid.

AMRC_SENDER_NOT_UNIQUE
The specified name could not be resolved to a unique sender because more than one sender object with that name exists.

AMRC_SENDER_USAGE_ERR
The specified sender service definition type was not valid for sending responses. To be valid for sending a response, a sender service must not

have a repository definition, must have been specified as a response service when receiving a previous request message and must not have been used for any purpose other than sending responses.

AMRC_SERVICE_ALREADY_CLOSED
The specified (sender, receiver, distribution list, publisher or subscriber) service was already closed.

AMRC_SERVICE_ALREADY_OPEN
The specified (sender, receiver, distribution list, publisher or subscriber) service was already open.

AMRC_SERVICE_FULL
The specified request could not be performed because the service in the underlying transport has reached its maximum message limit.

AMRC_SERVICE_HANDLE_ERR
The service handle specified for a sender, receiver, distribution list, publisher, or subscriber was not valid.

AMRC_SERVICE_NOT_FOUND
The specified (sender, receiver, distribution list, publisher, or subscriber) service was not found, so the request was not carried out.

AMRC_SERVICE_NOT_OPEN
The request failed because the specified (sender, receiver, distribution list, publisher or subscriber) service was not open.

AMRC_SESSION_ALREADY_CLOSED
The session was already closed (or terminated).

AMRC_SESSION_ALREADY_OPEN
The session was already open (or initialized).

AMRC_SESSION_HANDLE_ERR
The specified session handle was not valid.

AMRC_SESSION_NOT_OPEN
The request failed because the session was not open.

AMRC_SUBSCRIBER_NOT_UNIQUE
The specified name could not be resolved to a unique subscriber because more than one subscriber object with that name exists.

AMRC_TRANSPORT_ERR
An error was reported by the underlying (MQSeries) message transport layer. The message transport reason code can be obtained by the secondary reason code value returned from a 'GetLastError' request for the AMI object concerned. For more information, see "Common causes of problems" on page 297.

AMRC_TRANSPORT_LIBRARY_ERR
An error occurred loading the transport library.

AMRC_TRANSPORT_NOT_AVAILABLE
The underlying transport layer is not available.

AMRC_UNEXPECTED_ERR
An unexpected error occurred.

AMRC_WAIT_TIME_ERR
The specified wait-time value was not valid.

AMRC_WAIT_TIME_PTR_ERR
The specified wait time pointer was not valid.

AMRC_WAIT_TIME_READ_ONLY
An attempt was made to set the wait time in a policy object for which the wait-time was read-only.

Reason code (failed)

Appendix B. Constants

This appendix lists the values of the named constants used by the functions described in this manual. For information about MQSeries constants not in this list, see the *MQSeries Application Programming Reference* manual and the *MQSeries Programmable System Management* manual.

The constants are grouped according to the parameter or field to which they relate. Names of the constants in a group begin with a common prefix of the form AMxxxx_, where xxxx represents a string of 0 through 4 characters that indicates the nature of the values defined in that group. Within each group, constants are listed in numeric (or alphabetic) order.

Character strings are shown delimited by double quotation marks; the quotation marks are not part of the value.

AMB (Boolean constants)

AMB_FALSE	0L
AMB_TRUE	1L

AMBRW (Browse constants)

AMBRW_UNLOCK	1L
AMBRW_LOCK	2L
AMBRW_FIRST	4L
AMBRW_NEXT	8L
AMBRW_CURRENT	16L
AMBRW_RECEIVE_CURRENT	32L
AMBRW_DEFAULT	AMBRW_NEXT
AMBRW_LOCK_NEXT	(AMBRW_LOCK + AMBRW_NEXT)
AMBRW_LOCK_FIRST	(AMBRW_LOCK + AMBRW_FIRST)
AMBRW_LOCK_CURRENT	(AMBRW_LOCK + AMBRW_CURRENT)

AMCC (Completion codes)

AMCC_OK	0L
AMCC_WARNING	1L
AMCC_FAILED	2L

AMDEF (Service and policy definitions)

AMDEF_POL	"AMT.SYSTEM.POLICY"
AMDEF_PUB	"AMT.SYSTEM.PUBLISHER"
AMDEF_RCV	"AMT.SYSTEM.RECEIVER"
AMDEF_RSP_SND	"AMT.SYSTEM.RESPONSE.SENDER"
AMDEF_SND	"AMT.SYSTEM.SENDER"
AMDEF_SUB	"AMT.SYSTEM.SUBSCRIBER"
AMDEF_SYNC_POINT_POL	"AMT.SYSTEM.SYNCPOINT.POLICY"

AMDT (Definition type constants)

```
AMDT_UNDEFINED                   0L
AMDT_TEMP_DYNAMIC                2L
AMDT_DYNAMIC                     3L
AMDT_PREDEFINED                  4L
```

AMENC (Encoding constants)

```
AMENC_NORMAL                     0L
AMENC_REVERSED                   1L
AMENC_NORMAL_FLOAT_390           2L
AMENC_REVERSED_FLOAT_390         3L
AMENC_UNDEFINED                  4L
AMENC_NATIVE           AMENC_NORMAL  (UNIX)
AMENC_NATIVE         AMENC_REVERSED  (WIN32)
```

AMFMT (Format constants)

```
AMFMT_NONE                      "        "
AMFMT_RF_HEADER                 "MQHRF   "
AMFMT_STRING                    "MQSTR   "
```

AMGF and AMGRP (Group status constants)

```
AMGF_IN_GROUP                    1L
AMGF_FIRST                       2L
AMGF_LAST                        4L

AMGRP_MSG_NOT_IN_GROUP           0L
AMGRP_FIRST_MSG_IN_GROUP  ( AMGF_IN_GROUP | AMGF_FIRST )
AMGRP_MIDDLE_MSG_IN_GROUP   AMGF_IN_GROUP
AMGRP_LAST_MSG_IN_GROUP   ( AMGF_IN_GROUP | AMGF_LAST  )
AMGRP_ONLY_MSG_IN_GROUP   ( AMGF_IN_GROUP | AMGF_FIRST | AMGF_LAST )
```

AMH (Handle constants)

```
AMH_NULL_HANDLE       (AMHANDLE) 0L
AMH_INVALID_HANDLE    (AMHANDLE)-1L
```

AMLEN (String length constants)

```
AMLEN_NULL_TERM                 -1L
AMLEN_MAX_NAME_LENGTH          256L
```

AMPS (Publish/subscribe)

Publish/subscribe tag names

AMPS_COMMAND	"MQPSCommand"
AMPS_COMP_CODE	"MQPSCompCode"
AMPS_DELETE_OPTIONS	"MQPSDelOpts"
AMPS_ERROR_ID	"MQPSErrorId"
AMPS_ERROR_POS	"MQPSErrorPos"
AMPS_PARAMETER_ID	"MQPSParmId"
AMPS_PUBLICATION_OPTIONS	"MQPSPubOpts"
AMPS_TIMESTAMP	"MQPSPubTime"
AMPS_Q_MGR_NAME	"MQPSQMgrName"
AMPS_Q_NAME	"MQPSQName"
AMPS_REASON	"MQPSReason"
AMPS_REASON_TEXT	"MQPSReasonText"
AMPS_REGISTRATION_OPTIONS	"MQPSRegOpts"
AMPS_SEQUENCE_NUMBER	"MQPSSeqNum"
AMPS_STREAM_NAME	"MQPSStreamName"
AMPS_STRING_DATA	"MQPSStringData"
AMPS_TOPIC	"MQPSTopic"
AMPS_USER_ID	"MQPSUserId"

Publish/subscribe tag values

AMPS_ANONYMOUS	"Anon"
AMPS_CORREL_ID_AS_ID	"CorrelAsId"
AMPS_DEREGISTER_ALL	"DeregAll"
AMPS_DIRECT_REQUESTS	"DirectReq"
AMPS_INCLUDE_STREAM_NAME	"InclStreamName"
AMPS_INFORM_IF_RETAINED	"InformIfRet"
AMPS_LOCAL	"Local"
AMPS_NEW_PUBS_ONLY	"NewPubsOnly"
AMPS_PUB_ON_REQUEST_ONLY	"PubOnReqOnly"

Other publish/subscribe constants

AMPS_APPL_TYPE	"OPT_APP_GRP "
AMPS_MSG_TYPE	"OPT_MSG_TYPE "

AMRC (Reason codes)

Reason codes 500 to 505 are not applicable to the C and C++ programming languages.

AMRC_NONE	0
AMRC_UNEXPECTED_ERR	1
AMRC_INVALID_Q_NAME	2
AMRC_INVALID_SENDER_NAME	3
AMRC_INVALID_RECEIVER_NAME	4
AMRC_INVALID_PUBLISHER_NAME	5
AMRC_INVALID_SUBSCRIBER_NAME	6
AMRC_INVALID_POLICY_NAME	7
AMRC_INVALID_MSG_NAME	8
AMRC_INVALID_SESSION_NAME	9
AMRC_INVALID_DIST_LIST_NAME	10
AMRC_POLICY_HANDLE_ERR	11
AMRC_SERVICE_HANDLE_ERR	12
AMRC_MSG_HANDLE_ERR	13
AMRC_SESSION_HANDLE_ERR	14
AMRC_BROWSE_OPTIONS_ERR	15
AMRC_INSUFFICIENT_MEMORY	16
AMRC_WAIT_TIME_READ_ONLY	17
AMRC_SERVICE_NOT_FOUND	18
AMRC_MSG_NOT_FOUND	19
AMRC_POLICY_NOT_FOUND	20
AMRC_SENDER_NOT_UNIQUE	21
AMRC_RECEIVER_NOT_UNIQUE	22
AMRC_PUBLISHER_NOT_UNIQUE	23
AMRC_SUBSCRIBER_NOT_UNIQUE	24
AMRC_MSG_NOT_UNIQUE	25
AMRC_POLICY_NOT_UNIQUE	26
AMRC_DIST_LIST_NOT_UNIQUE	27
AMRC_RECEIVE_BUFF_PTR_ERR	28
AMRC_RECEIVE_BUFF_LEN_ERR	29
AMRC_SEND_DATA_PTR_ERR	30
AMRC_SEND_DATA_LEN_ERR	31
AMRC_INVALID_IF_SERVICE_OPEN	32
AMRC_SERVICE_ALREADY_OPEN	33
AMRC_DATA_SOURCE_NOT_UNIQUE	34
AMRC_NO_MSG_AVAILABLE	35
AMRC_SESSION_ALREADY_OPEN	36
AMRC_SESSION_ALREADY_CLOSED	37
AMRC_ELEM_NOT_FOUND	38
AMRC_ELEM_COUNT_PTR_ERR	39

```
AMRC_ELEM_NAME_PTR_ERR          40
AMRC_ELEM_NAME_LEN_ERR          41
AMRC_ELEM_INDEX_ERR             42
AMRC_ELEM_PTR_ERR               43
AMRC_ELEM_STRUC_ERR             44
AMRC_ELEM_STRUC_NAME_ERR        45
AMRC_ELEM_STRUC_VALUE_ERR       46
AMRC_ELEM_STRUC_NAME_BUFF_ERR   47
AMRC_ELEM_STRUC_VALUE_BUFF_ERR  48
AMRC_TRANSPORT_ERR              49

AMRC_TRANSPORT_WARNING          50
AMRC_ENCODING_INCOMPLETE        51
AMRC_ENCODING_MIXED             52
AMRC_ENCODING_ERR               53
AMRC_BEGIN_INVALID              54
AMRC_NO_REPLY_TO_INFO           55
AMRC_SERVICE_ALREADY_CLOSED     56
AMRC_SESSION_NOT_OPEN           57
AMRC_DIST_LIST_INDEX_ERR        58
AMRC_WAIT_TIME_ERR              59

AMRC_SERVICE_NOT_OPEN           60
AMRC_HEADER_TRUNCATED           61
AMRC_HEADER_INVALID             62
AMRC_DATA_LEN_ERR               63
AMRC_BACKOUT_REQUEUE_ERR        64
AMRC_BACKOUT_LIMIT_ERR          65
AMRC_COMMAND_ALREADY_EXISTS     66
AMRC_UNEXPECTED_RECEIVE_ERR     67
AMRC_UNEXPECTED_SEND_ERR        68

AMRC_SENDER_USAGE_ERR           70
AMRC_MSG_TRUNCATED              71
AMRC_CLOSE_SESSION_ERR          72
AMRC_READ_OFFSET_ERR            73
AMRC_RFH_ALREADY_EXISTS         74
AMRC_GROUP_STATUS_ERR           75
AMRC_MSG_ID_LEN_ERR             76
AMRC_MSG_ID_PTR_ERR             77
AMRC_MSG_ID_BUFF_LEN_ERR        78
AMRC_MSG_ID_BUFF_PTR_ERR        79

AMRC_MSG_ID_LEN_PTR_ERR         80
AMRC_CORREL_ID_LEN_ERR          81
AMRC_CORREL_ID_PTR_ERR          82
AMRC_CORREL_ID_BUFF_LEN_ERR     83
AMRC_CORREL_ID_BUFF_PTR_ERR     84
AMRC_CORREL_ID_LEN_PTR_ERR      85
AMRC_FORMAT_LEN_ERR             86
AMRC_FORMAT_PTR_ERR             87
AMRC_FORMAT_BUFF_PTR_ERR        88
AMRC_FORMAT_LEN_PTR_ERR         89
```

AMRC_FORMAT_BUFF_LEN_ERR	90
AMRC_NAME_BUFF_PTR_ERR	91
AMRC_NAME_LEN_PTR_ERR	92
AMRC_NAME_BUFF_LEN_ERR	93
AMRC_Q_NAME_LEN_ERR	94
AMRC_Q_NAME_PTR_ERR	95
AMRC_Q_NAME_BUFF_PTR_ERR	96
AMRC_Q_NAME_LEN_PTR_ERR	97
AMRC_Q_NAME_BUFF_LEN_ERR	98
AMRC_WAIT_TIME_PTR_ERR	99

AMRC_CCSID_PTR_ERR	100
AMRC_ENCODING_PTR_ERR	101
AMRC_DEFN_TYPE_PTR_ERR	102
AMRC_CCSID_ERR	103
AMRC_DATA_LEN_PTR_ERR	104
AMRC_GROUP_STATUS_PTR_ERR	105
AMRC_DATA_OFFSET_PTR_ERR	106
AMRC_RESP_SENDER_HANDLE_ERR	107
AMRC_RESP_RECEIVER_HANDLE_ERR	108
AMRC_NOT_AUTHORIZED	109

AMRC_TRANSPORT_NOT_AVAILABLE	110
AMRC_BACKED_OUT	111
AMRC_INCOMPLETE_GROUP	112
AMRC_SEND_DISABLED	113
AMRC_SERVICE_FULL	114
AMRC_NOT_CONVERTED	115
AMRC_RECEIVE_DISABLED	116
AMRC_GROUP_BACKOUT_LIMIT_ERR	117
AMRC_SENDER_COUNT_PTR_ERR	118
AMRC_MULTIPLE_REASONS	119

AMRC_NO_RESP_SERVICE	120
AMRC_DATA_PTR_ERR	121
AMRC_DATA_BUFF_LEN_ERR	122
AMRC_DATA_BUFF_PTR_ERR	123
AMRC_DEFN_TYPE_ERR	124

AMRC_INVALID_TRACE_LEVEL	400
AMRC_CONN_NAME_NOT_FOUND	401
AMRC_HOST_FILE_NOT_FOUND	402
AMRC_HOST_FILENAME_ERR	403
AMRC_HOST_FILE_ERR	404
AMRC_POLICY_NOT_IN_REPOS	405
AMRC_SENDER_NOT_IN_REPOS	406
AMRC_RECEIVER_NOT_IN_REPOS	407
AMRC_DIST_LIST_NOT_IN_REPOS	408
AMRC_PUBLISHER_NOT_IN_REPOS	409
AMRC_SUBSCRIBER_NOT_IN_REPOS	410

AMRC_RESERVED_NAME_IN_REPOS	411
AMRC_REPOS_FILENAME_ERR	414
AMRC_REPOS_WARNING	415
AMRC_REPOS_ERR	416
AMRC_REPOS_NOT_FOUND	418
AMRC_TRANSPORT_LIBRARY_ERR	419
AMRC_JAVA_FIELD_ERR	500
AMRC_JAVA_METHOD_ERR	501
AMRC_JAVA_CLASS_ERR	502
AMRC_JAVA_JNI_ERR	503
AMRC_JAVA_CREATE_ERR	504
AMRC_JAVA_NULL_PARM_ERR	505

AMSD (System default names and handles)

Default names

AMSD_POL	"SYSTEM.DEFAULT.POLICY"
AMSD_PUB	"SYSTEM.DEFAULT.PUBLISHER"
AMSD_PUB_SND	"SYSTEM.DEFAULT.PUBLISHER"
AMSD_RCV	"SYSTEM.DEFAULT.RECEIVER"
AMSD_RCV_MSG	"SYSTEM.DEFAULT.RECEIVE.MESSAGE"
AMSD_RSP_SND	"SYSTEM.DEFAULT.RESPONSE.SENDER"
AMSD_SND	"SYSTEM.DEFAULT.SENDER"
AMSD_SND_MSG	"SYSTEM.DEFAULT.SEND.MESSAGE"
AMSD_SESSION_NAME	"SYSTEM.DEFAULT.SESSION"
AMSD_SUB	"SYSTEM.DEFAULT.SUBSCRIBER"
AMSD_SUB_SND	"SYSTEM.DEFAULT.SUBSCRIBER"
AMSD_SUB_RCV	"SYSTEM.DEFAULT.SUBSCRIBER.RECEIVER"
AMSD_SYNC_POINT_POL	"SYSTEM.DEFAULT.SYNCPOINT.POLICY"

Default handles

AMSD_RSP_SND_HANDLE	(AMHSND)-5L
AMSD_RCV_HANDLE	(AMHRCV)-6L
AMSD_POL_HANDLE	(AMHPOL)-7L
AMSD_SYNC_POINT_POL_HANDLE	(AMHPOL)-8L
AMSD_SND_MSG_HANDLE	(AMHMSG)-9L
AMSD_RCV_MSG_HANDLE	(AMHMSG)-10L

AMWT (Wait time constant)

AMWT_UNLIMITED	-1L

Appendix C. Notices

This information was developed for products and services offered in the United States. IBM may not offer the products, services, or features discussed in this information in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this information. The furnishing of this information does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Trademarks

Part 7. Glossary and index

Glossary of terms and abbreviations

This glossary defines terms and abbreviations used in this book. If you do not find the term you are looking for, see the Index or the *IBM Dictionary of Computing*, New York: McGraw-Hill, 1994.

C

Connection. An AMI connection maps a logical queue manager name in a policy to a real queue manager name. This allows applications running on different nodes to use the same policy to connect to different queue managers.

Correlation identifier. This is used as a key to a message, for example to correlate a response message with a request message. The AMI normally sets this in a response message by copying the message identifier from the request message. See also *request/response* and *selection message*.

D

Datagram. The simplest message that MQSeries supports. Also known as *send-and-forget*. This type of message does not require a reply. Compare with *request/response*.

Distribution list. An AMI service. It contains a list of sender services, enabling a message to be sent to multiple destinations in one operation.

L

Local host file. Defines the mapping from a logical connection name to a real MQSeries queue manager on the local machine.

M

Message. A message defines what is sent from one program to another in an AMI application. See also *service* and *policy*.

Message descriptor (MQMD). Control information describing the message format and properties that is carried as part of an MQSeries message.

Message identifier. An identifier for the message. It is usually unique, and typically it is generated by the message transport (MQSeries).

Message object. An AMI object. It contains attributes of the message, such as the message identifier and correlation identifier, and options that are used when sending or receiving the message (most of which come from the policy definition). It can also contain the message data.

Message queue. See *queue*.

Message queue interface (MQI). The programming interface provided by MQSeries queue managers. It allows application programs to access message queuing services. The AMI provides a simpler interface to these services.

MQRFH header. Header added to an MQSeries message to carry control information, typically for use by a broker (for example, in a publish/subscribe system).

P

Point-to-point. Style of messaging application in which the sending application knows the destination of the message. Compare with *publish/subscribe*.

Policy. A policy defines how a message is sent in an AMI application. It encapsulates many of the options available in the MQI. Its definition can be stored in a repository. See also *service*.

Publish/subscribe. Style of messaging application in which the providers of information (publishers) are decoupled from the consumers of that information (subscribers) using a broker. Compare with *point-to-point*. See also *topic*.

Publisher. (1) An AMI service. It contains a sender service where the destination is a publish/subscribe broker. (2) An application that makes information about a specified topic available to a broker in a publish/subscribe system.

Q

Queue. An MQSeries object. Message queuing applications can put messages on, and get messages from, a queue. A queue is owned and maintained by a queue manager. Local queues can contain a list of messages waiting to be processed. Queues of other types cannot contain messages: they point to other queues, or can be used as models for dynamic queues.

Queue manager. A system program that provides queuing services to applications. It provides an application programming interface (the MQI) so that programs can access messages on the queues that the queue manager owns.

R

Receiver. An AMI service. It represents a source (such as an MQSeries queue) from which messages are received. Its definition is stored in a repository as a service point.

Repository. A repository provides definitions for services and policies. If the name of a service or policy is not found in the repository, or an AMI application does not have a repository, the definitions built into the AMI are used. See also *repository file*.

Repository file. File that stores repository definitions in XML (Extensible Markup Language) format.

Request/response. Type of messaging application in which a request message is used to request a response from another application. Compare with *datagram*. See also *response sender* and *selection message*.

Response sender. A special type of sender service that is used to send a response to a request message. It must use the definition built into the AMI, so it must not be defined in the repository.

S

Selection message. A message object that is used to selectively receive a message by specifying its correlation identifier. Used in request/response messaging to correlate a response message with its request message.

Send-and-forget. See *datagram*.

Sender. An AMI service. It represents a destination (such as an MQSeries queue) to which messages are sent. Its definition is stored in a repository as a service point.

Service. A service defines where a message is sent in an AMI application. Senders, receivers, distribution lists, publishers, and subscribers are all types of service. Their definitions can be stored in a repository. See also *policy*.

Service point. The definition in a repository of a sender or receiver service.

Session. An AMI object. It creates and manages all other AMI objects (message, service, policy and connection objects), and it provides the scope for a unit of work when transactional processing is used.

Subscriber. (1) An AMI service. It contains a sender service to send subscribe and unsubscribe messages to a publish/subscribe broker, and a receiver service to receive publications from the broker. (2) An application that requests information about a specified topic from a publish/subscribe broker.

T

Topic. A character string that describes the nature of the data that is being published in a publish/subscribe system.

Index

Sending your comments to IBM

MQSeries®

Application Messaging Interface

SC34-5604-00

If you especially like or dislike anything about this book, please use one of the methods listed below to send your comments to IBM.

Feel free to comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book. Please limit your comments to the information in this book and the way in which the information is presented.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate, without incurring any obligation to you.

You can send your comments to IBM in any of the following ways:

- By mail, use the Readers' Comment Form
- By fax:
  - From outside the U.K., after your international access code use 44 1962 870229
  - From within the U.K., use 01962 870229
- Electronically, use the appropriate network ID:
  - IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
  - IBMLink: HURSLEY(IDRCF)
  - Internet: idrcf@hursley.ibm.com

Whichever you use, ensure that you include:

- The publication number and title
- The page number or topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

Readers' Comments

MQSeries®

Application Messaging Interface

SC34-5604-00

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Be sure to print your name and address below if you would like a reply.

Name	Address
Company or Organization	
Telephone	Email

MQSeries®

MQSeries Application Messaging Interface SC34-5604-00

You can send your comments POST FREE on this form from any one of these countries:

Australia	Finland	Iceland	Netherlands	Singapore	United States of America
Belgium	France	Israel	New Zealand	Spain	
Bermuda	Germany	Italy	Norway	Sweden	
Cyprus	Greece	Luxembourg	Portugal	Switzerland	
Denmark	Hong Kong	Monaco	Republic of Ireland	United Arab Emirates	

If your country is not listed here, your local IBM representative will be pleased to forward your comments to us. Or you can pay the postage and send the form direct to IBM (this includes mailing in the U.K.).

1 Cut along this line

2 Fold along this line

By air mail
Par avion

IBRS/CCRI NUMBER: PHQ - D/1348/SO

NE PAS AFFRANCHIR

NO STAMP REQUIRED

REPONSE PAYEE
GRANDE-BRETAGNE

IBM United Kingdom Laboratories
Information Development Department (MP095)
Hursley Park,
WINCHESTER, Hants
SO21 2ZZ United Kingdom

3 Fold along this line

From: Name ______________________
Company or Organization ______________________
Address ______________________
____
EMAIL ______________________
Telephone ______________________

1 Cut along this line

4 Fasten here with adhesive tape