Programmer's Manual Version 6

A highly scalable, open architecture, internet messaging system that runs on Windows and Linux platforms.

ISBN: **962-8137-34-4**
Document ID: **IEMS6MQAPIM001**
Date of Publication: **April, 2002**

CONTENTS

PREFACE

The Internet Exchange Messaging Server (IEMS) 6 supports a C++ Message Queue Application Programming Interface (MQAPI). This permits submission and retrieval of messages to and from the IEMS Message Queue.

This manual is intended for C++ programmers who wish to create third party software for IEMS. It also discusses the steps on how the MQAPI connect to the system. It describes how to use the MQAPI to write third party applications that submit and fetch messages to and from the IEMS Message Queue.

This document is organized as follows:

Overview	Background information on the Internet Exchange Messaging Server
Introduction	Material on the MQ API and how it works with the other IEMS components
API Class Definitions	Class and methods which compose the IEMS MQ API
API Function Reference	Detailed discussions of methods and functions of the MQ API
How to Use the MQAPI	How to use the MQ API with third party applications. Discusses the step by step procedure on how to add external modules and stops on how to add other input-output combinations
Error Codes	Codes generated by the API functions when error occur
Appendix A	Sample application program

OVERVIEW

The Internet Exchange Messaging Server (IEMS) is an open architecture, fully featured messaging system that complies with Internet standards to ensure smooth and reliable transmission of messages. At the heart of IEMS is a number of components which work together to send and receive messages (see Figure 1 on page 5).

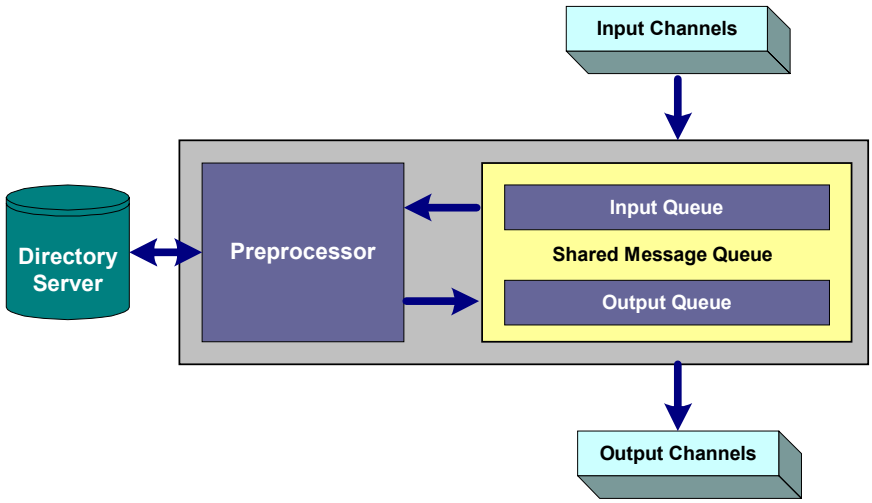

Figure 1: IEMS System Architecture

The IEMS input channels receive messages from the different applications supported by IEMS. For example, the CCOUT channel receives messages from Lotus cc:Mail.

Aside from CCOUT, IEMS also supports the following input channels:

- **BSMTPIN** - receives messages from the Internet transmitted via POP3 connection
- **DL** - receives messages sent to distribution lists
- **LOCALOUT** - channel used by the LMDA to forward messages via the Mailsort module
- **NOTESOUT**- imports messages from the Notes environment.
- **SMTPD** - receives messages from the Internet via standard SMTP
- **WEB MAIL CLIENT** - web based user agent that connects users to the local Message Store.

The Input channels submit messages received to the MQ Server. The MQ server forwards these messages to the Preprocessor that performs address resolution/expansion, virus scanning, spam checks, disclaimer insertion.

The MQ Server manages the Shared Message Queue - the centralized mail holding area that stores messages (physical) awaiting delivery in holding areas called queues. It also creates and manages the Preprocessor and MQDBStr message databases.

The Preprocessor accesses the Preprocessor database to perform address resolution/expansion for the messages. Afterwhich, it facilitates virus scanning, spam detection and disclaimer insertion among others on messages.

The Directory server, which uses the LDAP access protocol, stores and manages information about users, groups, mailing lists, alias processing and mail routing. The Preprocessor accesses this information to determine recipient addresses/routing information for each message.

While the input channels submit messages to the MQ Server for preprocessing, the output channels fetch messages from the MQ Server to deliver them to their intended recipients. The following output channels are supported by IEMS:

- **BSMTPOUT** - delivers messages to its intended recipient on the other end of the BSMTP Tunnel
- **CCIN** - exports messages to the cc:Mail environment
- **DLOUT** - delivers messages intended for Distribution List members
- **LOCAL** - deliver messages to the Lotus Message Store
- **NOTESIN** - delivers messages to the Lotus Notes environment
- **SMTPC** - receives messages destined for the Internet from the IEMS MTA and routes them to other mail servers on the Internet

INTRODUCTION

The Message Queue is the centralized mail repository that stores messages (physical) awaiting delivery in holding areas called queues. The queues are classified into two groups: the input and output queues.

The input channels submit messages to the input queues for preprocessing while the output channels fetch preprocessed messages from the output queues to deliver the messages to their intended recipients (see Figure 2 on page 7).

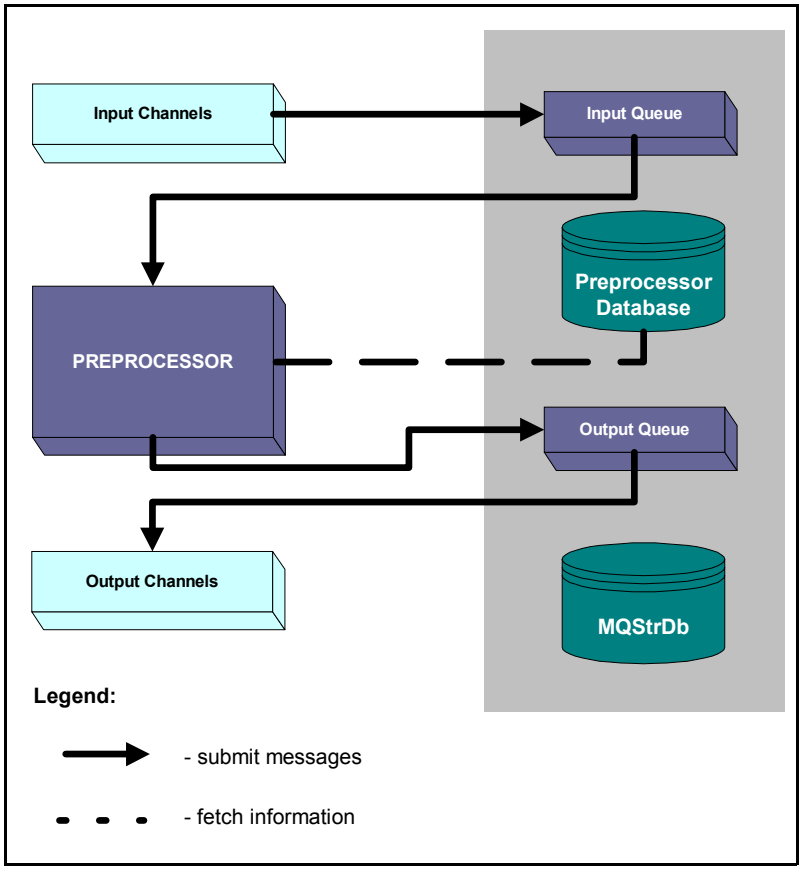

Figure 2: Message Flow

Software developers can create applications that submit and fetch messages to the Message Queue via the MQAPI.

The MQAPI is a set of functions/routines that enable access to the Message Queue. Implemented as a dynamic link library (API_MQ.dll for Windows and

libmq.so for Linux), the MQ API consists of functions that perform the following operations:

- Open a connection to the MQ Server
- Insert a message into an input channel
- Fetch a message from an output channel
- Obtain the path where the message was physically stored
- Delete fetched message from the output channel
- Close an open connection to the MQ Server

Upon receipt of messages, an input channel opens a connection to the MQ server (see Figure 3 on page 8) and submits its messages to its respective input queue in the Message Queue Server.

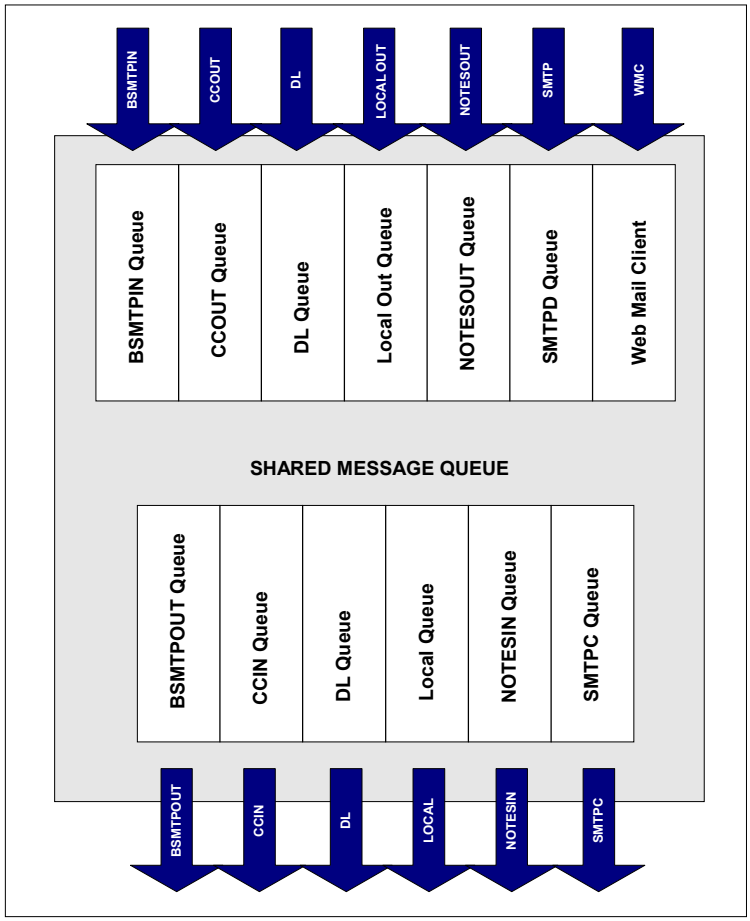

Figure 3: Channel I/O Mapping

When a message is submitted to the Message Queue Server, the Message Queue Server, in turn, submits it to the Preprocessor for any potential preprocessing. This is done by creating a new entry in the Preprocessor database,

MQPreprDB.db. This new entry, which is indexed by a unique message identifier or qid, consists of source and destination channel data, message envelope information and a reference to the file containing the RFC822 message.

Once the message is fetched by the Preprocessor (see Figure 4 on page 9), the Preprocessor will carry out three different tasks: 1) Envelope preprocesssing, 2) directory lookup, 3) calls to external modules (configurable). The third task is further divided into two.

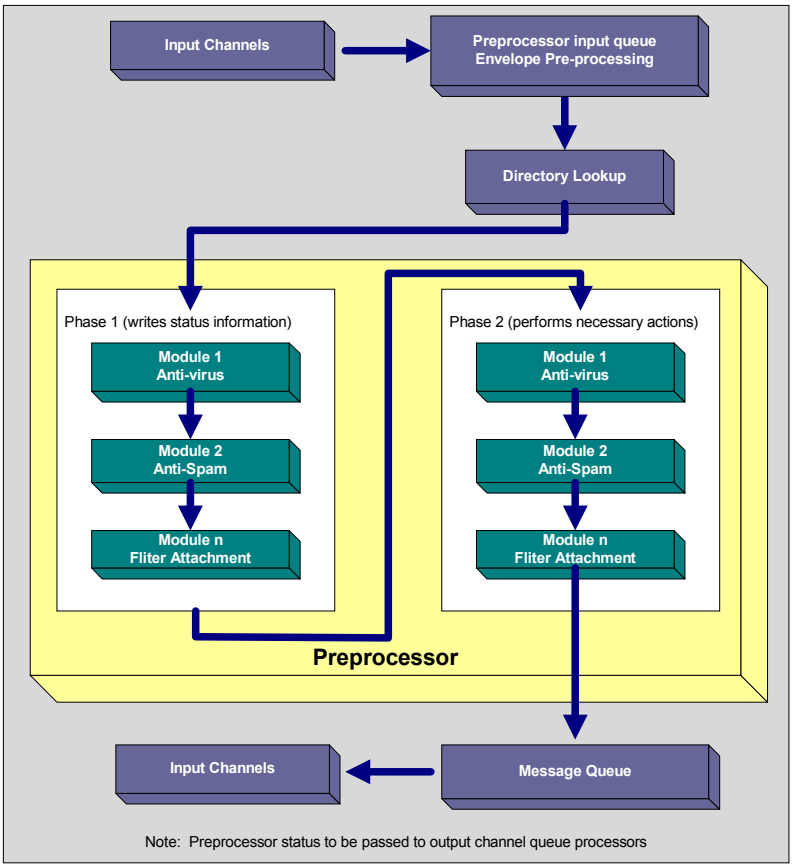

Figure 4: Tasks Performed by the Preprocessor

Envelope Preprocessing & Directory Lookup Stage

In these stages, the Preprocessor enumerates all the addresses in the MQ envelope and performs directory lookup from the Internal LDAP database to expand recipient addresses, determine output channels associated with the address and resolve mail aliases. It then copies the resolved internal address to the MQ envelope. This internal address can either be an IMAP/POP3 mailbox path, cc:Mail or Notes address or any channel.

Calling of the Preprocessor Plug-ins Stage

After directory lookup-address expansion, the Preprocessor plug-ins (i.e. anti-virus, anti-spam) are called to perform their respective functions on the messages. Preprocessor plug-ins are implemented as either Windows DLL's or Linux shared libraries. Each module undergoes two phases. In the first phase, it runs its routine. In phase 2, it performs the configured action on the messages based on the results in phase 1.

For example, if the anti-virus module is installed and configured in the Preprocessor, this module will undergo two phases. In phase 1, it runs routine, scan messages for viruses and separates the messages which are virus infected. In the phase 2, it performs the action (i.e. forward a message, deleting of message or send notification to the recipient or postmaster) configured by the system on the virus infected. When phase 1 or phase 2 is through the Preprocessor executes the next module. Once finished, the Preprocessor returns the control to the MQ server.

The Preprocessor plug-ins are configured in the IEMTA.INI (Windows) or IEMS.CONF (Linux) configuration file. In configuring this file, it should be remembered that DLLs are loaded only during run-time. Thus, to load the Preprocessor plug-ins DLL in the Preprocessor, the LoadLibrary(), GetProcAddress() and FreeLibrary() functions should be used and (for Windows), the dlopen, dlsym and dlclose functions should be used (for Linux). These functions are located in the Preprocessor module.

The `LoadLibrary()` is used to load the DLLs dynamically during run-time.
The `GetProcAddress()` is used to map the function address in the required DLLs.
The `FreeLibrary()` is used to unload the DLL library.

The `dlopen` is used to load SO (Shared Object) during run-time.
The `dlsym` is used to map the function address in the required SO.
The `dlclose` is used to unload the SO library.

To simplify the process taken by the Preprocessor, each external module located in the Preprocessor (i.e. Anti-virus, SpamArchive, SpamDelete, etc.) is given a specific Channel Action Matrix. This Channel Action Matrix defines all the possible input and output channel combinations where the messages may flow. It also determines which Preprocessor plug-ins should be called for messages flowing through an input-output channel combination.

For example, if the system administrator decides to run the anti-virus module to scan messages from the SMTPD destined to the SMTPC, he will configured the channel action matrix by selecting the proper channel for the messages. The Channel Action Matrix is implemented as file that stores the relationship between channel trace and Preprocessor actions by maintaining a channel trace for each message. It records the name of the input & output channel where a particular message passed through. For example, a message received by the SMTPD will have a channel trace equal to SMTPD. Once the message is routed to SMTPC, the channel trace becomes SMTPD:SMTPC.

In configuring the channel action matrix (preproc.cfg) file. The format of the configuration file is as follows:

<Channel_Trace>=<UniqueIdentifier>

e.g.

1. SMTPD:SMTPC= Anti-Virus
2. SMTPD:SMTPC= SpamDelete

The first example means any message received by the SMTPD destined to SMTPC will undergo spam checks so Preprocessor will call the Anti-Spam plug in for the messages that flow through these channel. The messages that meet the criteria of Anti-spam will be deleted. The second example means the Preprocessor should run the anti-spam and the filter attachment module to process all messages flowing through SMTPD going to SMTPC (see Figure 5 on page 11).

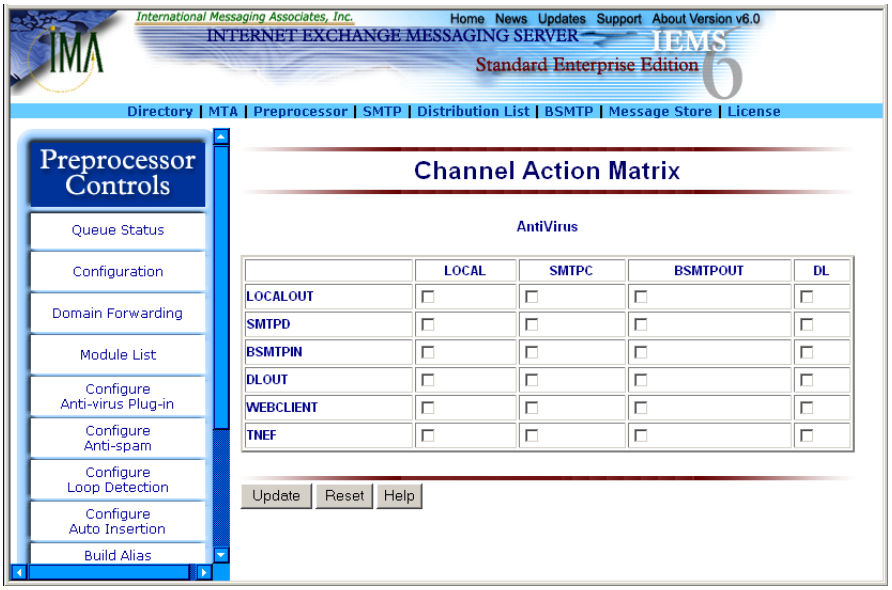

Figure 5: Channel Action Matrix

Note: *Once the Preprocessor is done with a particular message, it informs the Message Queue Server of the message's qid. The Message Queue Server then looks up the destination channels appropriate for the given qid in the Preprocessor database. Consequently, a new entry corresponding to the message will be created in the database of each such destination channel set by the Preprocessor.*

CHAPTER 1

API Class Definitions

Class cMQ

This class is used whenever a message will be submitted to or retrieved from the queue or channel. This includes the operations supported by the API.

Class Declaration

```
class MQCLASSDECL cMQ {
      int count;
      bool initial;
      char* appname;
      void* inMQ;
      void* outMQ;
      void* myEnv;
      cMessage* msg;
      cEnvHeader from;
      cEnvHeader to;
      char* path;
      bool bDelete;
      unsigned long idx;

public:

      cMQ();
      ~cMQ();
      int OpenMQChannel( char* appname, char* ldap );
      char* GetMsgDest( char* ext );
      char* GetPathName(unsigned long id, char* ext);
      cMessage* GetMsgEnv(unsigned long id);
      int PutMsg( cMessage* msg, char* inchannel );
      cMessage* GetMsg( char* outchannel );
      int DelMsg();
      int CloseMQChannel();
};
```

Methods Used

OpenMQChannel - opens a connection to an input queue in the MQ Server and performs channel I/O mapping.
PutMsg -submits the message into the specified input channel.
GetMsg -fetch the message into the output queue.
DelMsg - deletes the current message file in the Queue directory.
GetMsgDest -returns the full pathname where the message will be written or saved.
GetPathName - returns the full pathname of the message.
GetMsgEnv - retrieves the envelope information of the message.
CloseMQChannel- closes the connection to the MQ Server.

Class cMessage

This class is used to access the list of user information for the recipient "to" and the sender "from" envelope header. It also has access to the full path-name where the messages are located.

Class Declaration

```
class MQCLASSDECL cMessage {
    cEnvHeader* from;
    cEnvHeader* to;
    char* msgpath;

public:

    cMessage();
    ~cMessage();
    void setFrom( cEnvHeader* from );
    cEnvHeader* getFrom();
    void setTo( cEnvHeader* to );
    cEnvHeader* getTo();
    void setMsgpath( char* msgpath );
    char* getMsgpath();
};
```

Methods Used

setFrom - sets values for the user information of the sender ("from") envelope header..

getFrom - retrieves the values for the sender ("from") envelope header.

setTo - sets values for the user information of the recipient ("to") envelope header

getTo - retrieves the user information of the recipient ("to") envelope header.

setMsgpath - sets the full path where the message file is located.

getMsgpath - retrieves the exact location of the message file.

Class cEnvHeader

This class is used when a list of user information for the recipient **TO:** and the sender **FROM:** envelope header needs to be created.

Class Declaration

```
class MQCLASSDECL cEnvHeader{
    cList <cUserInfo*>userlist;
    cUserInfo * tmp;

public:
    cEnvHeader()
    ~cEnvHeader();
    int add(cUserInfo* user);
    void display();
    int dell(char*key);
    cUserInfo* get(char* key)
    cUserInfo *getFirst();
    cUserInfo *getNext();
};
```

Methods Used

add - adds a new user to the list of envelope headers .
del - deletes a new user from the list of envelope headers.
delAll - deletes all user information in the list of envelope headers.
get - retrieves the user information.
getFirst - retrieves the first record or user information in the list.
getNext - retrieves the next record or user information in the list.
getName - retrieves the name of the user .
getLan_addr - retrieves the email address of the user.

Class cUserInfo

This class is used for accessing user record which stores username and email address.

Class Declaration

```
class MQCLASSDECL cUserInfo {
    char* name;
    char* lan_addr;

public:

    cUserInfo();
    ~cUserInfo();
    void setName( char* name );
    char* getName();
    int setLan_addr( char* lan_addr );
    char* getLan_addr();
};
```

Methods Used

setName - sets the name of the user.
getName - retrieves the user's name.
setLan_addr - sets the email address of the user.
getLan_addr - retrieves the email address of the user.

Class cChannel

This class is used for adding and deleting channel in the MQAPI.

Class Declaration

```
class cChannel {
    char *iniFileName;
    char *szQueueFile;
    char *szTmpFile;
    char szLdapHost[ SZ_HOST ];

public:

    cChannel();
    ~cChannel();
    bool IsExist( char *channel );
    int Add(char *channel, char *application, char *type, char compatibil-
    ity);
    int Del( char *channel );
};
```

Methods Used

IsExist - checks if the channel exist in the configuration file.
Add - adds channel entry in the file.
Del -deletes channel entry in the file.

CHAPTER 2

API Function Reference

cMQ:: OpenMQChannel

Description:
Opens a connection to the MQ Server. It also performs channel I/O mapping.

Syntax:
int OpenMQChannel (char*appname, char*ldap);

Parameter(s):
appname -name of the application accessing the message queue.
ldap -the location of Directory Server machine.

Returns:
Returns 0 if no errors occured. Otherwise, returns a non-zero value. (Please refer to the MQAPI Error Codes.)

Note: *For an application to access the Message Queue, it must first open a connection to the MQ Server. Hence, an application must first call OpenMQChannel before it can submit or retrieve messages to or from the Message Queue.*

cMQ::PutMsg

Description:
Inserts a message into the specified input channel.

Syntax:
int PutMsg(cMessage*msg, char* inchannel);

Parameter(s):
msg - Actual message file to be stored in the queue. This includes the user information for the recipient ("to") and the sender ("from") envelope header, and the full path where the message is located
inchannel - Name of the Input queue (e.g. LOCALOUT, CCOUT)

Returns:
Returns 0 if no errors occurred. Otherwise, a non-zero value (please refer to the MQ API Error Codes).

Note: *This function should be invoked after the channel where the message will be inserted have been successfully opened.*

cMQ:: GetMsgPath

Description:
Returns the whole path name where the message will be written.

Syntax:
const char *GetMsgPath (char*ext);

Parameter(s):
ext - specifies the extension name of the file.

Returns:
Returns the whole path name where the message will be written.

Note: *Called after PutMsg.*

cMQ::GetMsg

Description:
Retrieves a message from an output channel.

Syntax:
cMessage*GetMsg (char* outchannel);

Parameter(s):
outchannel - specifies the output channel.

Returns:
Returns the retrieved contents of the from and to envelope headers and the full pathname where the message is located.

Note: *Should be called only after a channel has been successfully opened.*

cMQ::DelMsg

Description:
Deletes the message file in the Queue directory.

Syntax:
int DelMsg ();

Parameter(s):
none

Returns:
Returns zero if message file was successfully deleted otherwise a nonzero value is returned (please refer to the MQ API Error Codes).

Note: *Should be called only after GetMsg() method has been successfully called.*

cMQ:: CloseMQChannel

Description:
Closes the connection to the Message Queue Server.

Syntax:
int CloseMQChannel();

Parameter(s):
none.

Returns:
Returns 0 if no error and a non-zero if error occurs (please refer to the MQ API Error Codes).

Note: *Should only be called if there exist an open channel.*

cChannel:: IsExist

Description:
Checks if the channel exist in the file.

Syntax:
bool IsExist(char *channel);

Parameter(s):
channel - specifies the input/output channel.

Returns:
Returns true if the channel is already exist, if not false.

cChannel:: Add

Descripton:
Add a channel entry in the file.

Syntax:
int Add(char *channel, char * application, char *type, char compatibility);

Parameter(s):
channel - specifies the input/output channel.
application - specifies the application name or process name.
type - specifies channel type (e.g. "in" or "out")
compatibility - specifies compatibility mode. The value of compatibility is 'c' if compatible and NULL otherwise.

Returns:
Returns 0 if the successful and non-zero if not (please refer to the MQ API Error Codes).

cChannel:: Del

Description:
Deletes channel entry in the file.

Syntax:
int Del(char *channel);

Parameter(s):
channel - specifies the input/output channel.

Returns:
Returns 0 if the channel is deleted, if not a non-zero value is returned.

cMQ:: GetPathName

Description:
Returns the full pathname of the message

Syntax:
char*GetPathName(unsigned long id, char* ext)

Parameter(s):
id - unique message identification.
ext - specifies the extension name of the file.

Return(s):
Returns the full pathname of the message.

cMQ:: GetMsgEnv

Description:
Retrieves the envelope information of the message.

Syntax:
cMessage* GetMsgEnv(unsigned long id)

Parameter(s):
id - unique message identification.

Return(s):
Returns the envelope information of the message.

MQ API Program Flow

- Create an instance of the class cMQ to enable basic queue operations like submitting and fetching messages.

  Syntax:
  cMQ a;

- Call the OpenMQChannel method. This will open a connection to a specific channel in the MQ Server.

  a.OpenMQChannel(appName, ldap)

- Put or fetch a message from the Message Queue.

  To submit:
  cMessage message; /*set message envelope and contents*/
  a.PutMsg(&message, "localout");

  To fetch a message:
  Call the method GetMsg(), which returns the cMessage object. To fetch again the next message just invoke once more the GetMsg(); After calling GetMsg, DelMsg() method is invoked to delete the file in the Queue directory.

  cMessage*msg = a.GetMsg("local");
  a.DelMsg();

- Lastly, when all the queue operations are through, invoke the CloseMQChannel to close the connection to the MQ Server.

  a.CloseMQChannel()

Note: *You may create and add your own queue in IEMS. All you need to do is update the queue.cfg file and add Input/Output channels manually.*

CHAPTER 3

How To Use The MQAPI

This section aims to help you further understand how you can use the MQAPI to develop third party applications for IEMS. It contains information on what precisely should be done to build an application (e.g. channel processor) for IEMS. It demonstrates how to hook the created application to IEMS and tie the program to the IEMS general administration interface.

Prerequisites

System Requirements

Hardware

Pentium 133 or higher model microprocessor
64 MB RAM
200MB hard disk space for applications
1GB hard disk space for message store

Software (Linux)

Linux OS (Redhat, Caldera, VALinux, TurboLinux Server, Suse, Mandrake)
IEMS 5.1 for Linux
Compiler: gcc 2.91.66 or above

Software (Windows)

Windows 98/NT/ 2000
IEMS 5.1 for Windows
Compiler: Visual C++ 5.0 or above

MQ API Toolkit

The MQAPI Toolkit contains the MQAPI files. It is separately packaged, but freely available. It is important that this toolkit be successfully installed when creating a new application for IEMS. To download the most recent version of the toolkit, please see:

http://www.ima.com/iems/api.html

Given below is the directory structure of the MQAPI files.

For WINDOWS & Linux Directories Definition:

Table 1:

Windows	Linux	
toolkit\mqapi\ include	toolkit\mqapi\ include	All message queue header files (*.h)
toolkit\mqapi\lib	toolkit\mqapi\lib	All libraries
toolkit\sample	toolkit\sample	Source codes for sample application programs
toolkit\sam-ple\debug	toolkit\sample\debug	Sample application object codes in debug mode.
toolkit\sample\ release	toolkit\sam-ple\release	Sample application object codes in release mode.

Building Applications Using MQAPI

Header files

In using the MQAPI, the first consideration is the inclusion of the MQAPI header files in the application source code. These files define the basic structures, function prototypes, and return codes needed to use the MQAPI.

mqapi.h

mqapi.h (for Windows & Linux) is a file to be included in C++ programs to provide definitions for the Message Queue Interface to IEMS. To include the directory path, use the /I for (Windows) and -I for (Linux).

Note: See sample program in the Appendix A

API_MQ.lib or libmq.so

API_MQ.lib (for Windows) or libmq.so (for Linux) is the library that contains the definitions for the IEMS public entry points to the Message Queue.

To include the library file in your application, do these steps:

For Windows:

1. Using Visual C++, go to the Project Setting of the application (see Figure 6 on page 27).

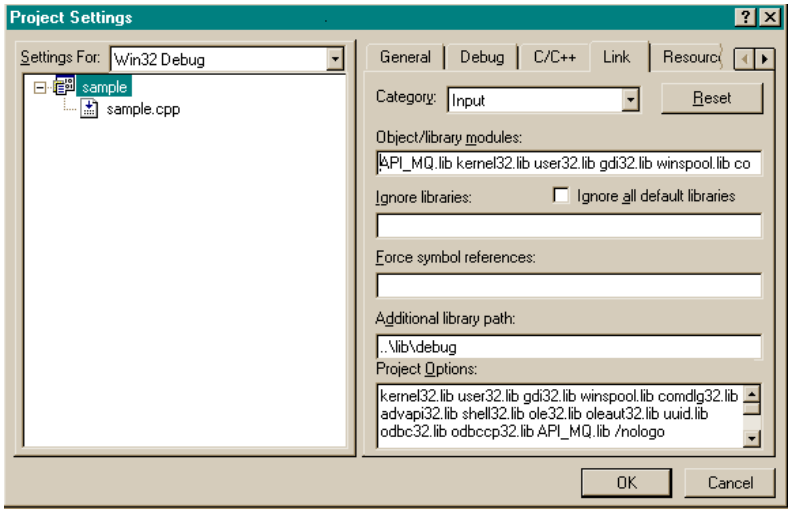

Figure 6: The Project Setting Dialog box

2. On the Project Setting dialog, select Link Tab.
3. In the Category option, select Input.
4. Go to the Object/library module textbox then add the library API_MQ.lib.
5. In the Additional/library path textbox, add the message queue library path.
6 Click ***OK*** button.

For Linux:
1. In the Linux makefile, add the library path using -L option.
2. Then inclide the library using -l option in the application program.

Adding Preprocessor Plug-ins In The Configuration File

The Preprocessor uses the prototype defined below to load the DLL Plug-ins:

DWORD Func(enum PHASE phase, QID qid, MQContext *mq, char *proc_file, char *szDestChannel)

The function uses the “phase number” (phase 1 or phase 2), “the QID of the message”, “the Message queue context” and “the name of the process file” as input parameters. The process file is assigned by the preprocessor module and it’s extension is “.pXX”, where “XX” refers to a number from 00 to 99.

During phase 1, the external function writes writes any status information into the process file and during phase 2, the actions on the message are performed based on the contents of the process file.

The Preprocessor plug-ins are configured in the IEMTA.INI (Windows) or IEMS.CONF (Linux). For your third party application to work within IEMS, the configuration file IEMTA.INI (Windows) or IEMS.CONF (Linux) must be modified to enable IEMS to recognize the newly added application. To modify, do the following:

1. Transfer the DLL (for Windows) or SO (Shared Objects for Linux) files to the Message Queue Directory.
2. Open the Configuration file.
3. Locate the Preprocessor label. See example below:

e.g.

```
[PreProcessor]
NumberOfModules =<N>
```

Where:
N can be 1 to 100

Note: *The maximum number of external modules that can be added to the Preprocessor is limited to 100 modules.*

After the NumberOfModules line comes the lines stating the configuration of the defined Preprocessor plug-ins. Each module is configured according to this syntax:

Module<N>=<DLL Name>,<FunctionName>,<UniqueIdentifier>
where:
<N> is the plug-in or module identification number
<DLL Name> is the name of the DLL
<FunctionName> is the DLL function entry point
<UniqueIdentifier> is the DLL unique identifier

e.g.

```
[PreProcessor]
NumberOfModules =2
Module0=anti_v.dll, anti_virus, Anti-Virus
Module1=anti_s.dll,anti_spam, SpamDelete
```

The above example, states the configuration of the anti-virus and anti-spam preprocessor modules.

To add another external module, first copy the DLLs (for Windows) or SO (Shared Objects for Linux) in the message queue directory. Type the module to be added (e.g. `Module 2=Filter.dll,FilterCheck,FilterAttachment`)in the configuration file. Then update the existing total number external modules (e.g. `NumberOfModules =3`) in the system.

e.g.

```
[PreProcessor]
NumberOfModules =3
Module0=anti_v.dll, anti_virus, Anti-Virus
Module1=anti_s.dll,anti_spam, SpamDelete
Module2=Filter.dll,FilterCheck,FilterAttachment
```

Creating The New Channel For Your Application

In order to create/add a new channel (input-output) for the created application and reflect the addition of these changes to the IEMS general administration interface, the programmer must perform the following:

Create a queue for the application

1. Load the IEMS LDAP application.

2. Create an instance of the object class channel.
   e.g. cChannel c;

3. Add a channel in "queue.cfg" file by calling the method
   Add(channel name, application name, type, mode).
   This supports the registering of the application to LDAP.
   e.g c.Add ("my", "smtpd", "in", NULL);

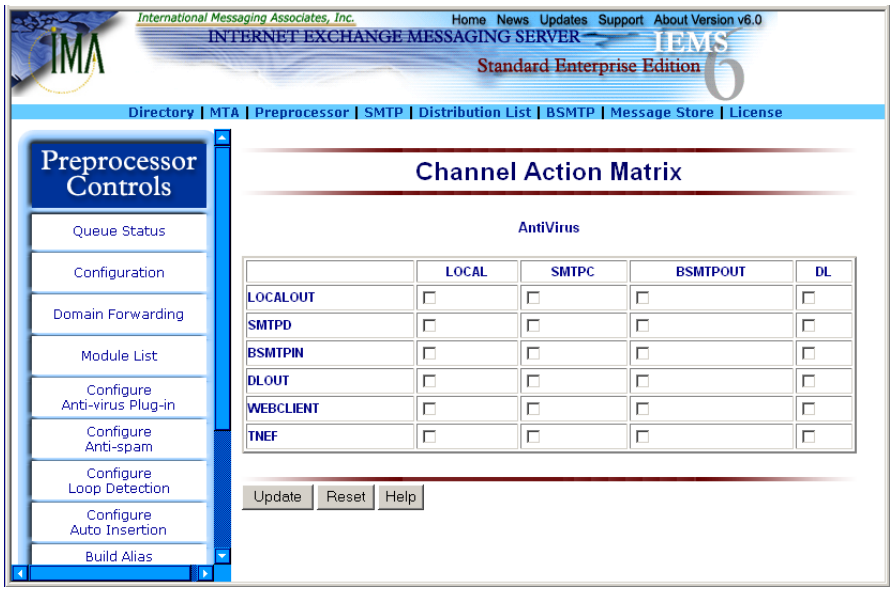

Figure 7: The Channel Action Matrix

Note: *You may also delete a channel by calling the method Del(channel name). This supports the unregistering of an application to LDAP. e.g. c.Del("my");*

After adding or deleting a channel, the file “queue.cfg” updates automatically. This file is located at `Program files/IMA/Internet Exchange Messaging Server 6.x` if IEMS is installed on Windows or in `/opt/iems` when IEMS is installed on Linux.

Conclusion

By incorporating an MQAPI, IEMS attests its open architecture and ensures its users interoperability and extensibility solutions for the future. It also presents an excellent opportunity for third party developers to create custom applications that would provide additional functionality to IEMS with flexibility.

APPENDIX A

TESTMQ.C Sample Program

The purpose of this program is to show how to insert and retrieve messages from the message queue.

```
#include "mqapi.h"
#include <iostream.h>
#define SUBMIT

int main() {

   /*set application name-test program name to "TESTMQ";*/
   char appname[] = "TESTMQ";

   /*set machine name of the LDAP server to "jasper.ima.com"*/
   char ldap[] = "jasper.ima.com";

   /*create an instance of class cMQ, name it a*/
   cMQ a;


#ifdef SUBMIT                        /* to submit messages*/
   cUserInfo user1, user2,user3;      /* create 3 instances of cUserInfo
                                         name it user1, user2, user3*/
   user1.setLan_addr("minnie@jasper.ima.com");  /*create test data*/
   user2.setLan_addr("marielle@jasper.ima.com");
   user3.setLan_addr("postmaster@jasper.ima.com");

/*create an object FROM*/
   /*create an instance of class cEnvHeader named from*/
   cEnvHeader* from = new cEnvHeader();

   /*assign the value of user1 to from's add property*/
   from->add( &user1 );


/*create an object TO*/
   /*create another instance of class cEnvHeader named to*/
   cEnvHeader* to = new cEnvHeader();

   /*assign the value of user2 to to's add property*/
   to->add( &user2 );

   /*append the value of user3 to to's add property*/
   to->add( &user3 );
```

```
   /* The code above simulates the envelope information of a message. From
      here, the envelope looks like:

      From: minnie@jasper.ima.com
      To  : marielle@jasper.ima.com
            postmaster@jasper.ima.com
    */

/*create an object of message*/
   /*create an instance of cMessage named msg*/
   cMessage msg;

   /*assign the value of from to the setFrom property of msg */
   msg.setFrom( from );

   /*assign the value of to to the setTo property of msg*/
   msg.setTo( to );


   /*error checking*/
   if(!a.OpenMQChannel( channel, appname, ldap ))
      cout<<"result = "<<"OK"<<"\n";

   /*insert msg to the local channel of the message queue*/
   a.PutMsg(&msg, "localout");

   /*assign the directory location of the inserted message to the
     setMsgpath property of msg*/
   msg.setMsgpath( ( char* )a.GetMsgPath( ".msg" ) );

   /*close the local channel of the Message Queue */
   a.CloseMQChannel();
#endif  /* SUBMIT */


#ifdef FETCH
   if(!a.OpenMQChannel( channel, appname, ldap ))
      cout<<"result = "<<"OK"<<"\n";

   /*create two instances of cMessage named msg2, msg3*/
   cMessage*msg2, *msg3;

   /*retrieve message from local output queue and assign value to msg2*/
   msg2 = a.GetMsg("local");

   /*error checking, if there is no message, return true*/
   if( msg2 == NULL )
      return(1);

   /*error checking to determine if from property of message is not null*/
   if( msg2->getFrom() == NULL )
      return(1);

   /*if message From: property has contents, display contents on screen */
```

```
    cout<<"from: "<<msg2->getFrom()->getFirst()->getLan_addr()<<"\n";


    /*error checking to determine if To property of message is null */
    if( msg2->getTo() == NULL )
      return(1);

    /*create an instance of cUserInfo p*/
    cUserInfo * p;

    /*get To: addresses and display them on screen */
    for( p=msg2->getTo()->getFirst(); p=msg2->getTo()->getNext(); ){
      cout<<"to:"<<p->getLan_addr()<<"\n";
    }

    /* Delete the message from the Queue directory */
    a.DelMsg();

    /*retrieve message from Message queue and assign value to msg3*/
    msg3 = a.GetMsg("local");

    /* error checking if msg is null return true */
    if( msg3 == NULL )
      return(1);

    /*error checking if message property From: is null return true*/
    if( msg3->getFrom() == NULL )
      return(1);

    /*display message property From: contents*/
    cout<<"from: "<<msg3->getFrom()->getFirst()->getLan_addr()<<"\n";

    /*error checking if message property To: is null, return true*/
    if( msg3->getTo() == NULL )
      return(1);

    /*get all contents of the To: property and display on screen*/
    for( p=msg3->getTo()->getFirst(); p=msg3->getTo()->getNext(); )
      cout<<"to:"<<p->getLan_addr()<<"\n";

    /* Delete the message from the Queue directory */
    a.DelMsg();

    /*close local channel in the MQ Server*/
    a.CloseMQChannel();
    #endif /* FETCH */
    return(0);
  }
```

APPENDIX B

MQ API Error Codes

Error Code	Value	Description
NO_ERR	0x00	No error
ERR_MALLOC	0x01	No available memory
ERR_NOTFOUND	0x02	Email address to delete is not found
ERR_NOENTRY	0x03	No available user information
ERR_EMAILADDRFMT	0x04	Invalid email address format
ERR_INVALIDCHANNELL	0x05	Invalid channel entry
ERR_MQINIT	0x06	MQ Initialization failure
ERR_OPENMQCHANNEL	0x07	Error in opening MQ channel
ERR_CLOSEMQCHANNEL	0x08	Error in closing MQ channel
ERR_CREATEMQENTRY	0x09	Error in allocating memory for new MQ entry
ERR_CREATEENV	0x0A	Error in allocating memory for new envelope data
ERR_NOFROM	0x0B	No FROM/sender data
ERR_NOTO	0x0C	No To/recipient data
ERR_NOMSGPATH	0x0D	Unable to retrieve message path
ERR_FILE	0x0E	Error in file manipulation
ERR_INSERTMQCHANNEL	0x0F	Unable to add channel entry to the MQ
ERR_PUTMQENVELOPE	0x10	Unable to store envelope information to the MQ
ERR_CLOSEMQENTRY	0x11	Unable to submit MQ entry to the preprocessor
ERR_REGISTERMODULE	0x12	Unable to register the application to LDAP
ERR_PARAM	0x13	Invalid input parameters
ERR_DELMSG	0x14	Error in deleting message file in the MQ
ERR_UNREGISTERMODULE	0x15	Unable to unregister the application to LDAP

APPENDIX C

License Agreement

THIS AGREEMENT SETS FORTH THE TERMS AND CONDITIONS UNDER WHICH THE SOFTWARE KNOWN AS IEMS API TOOLKIT WILL BE LICENSED BY INTERNATIONAL MESSAGING ASSOCIATES TO YOU, AND BY WHICH DERIVATIVE WORKS OF THE OPEN SOURCE AS PROVIDED IN THE IEMS API TOOLKIT WILL BE LICENSED BY YOU TO IMA. PLEASE READ THE FOLLOWING LICENSE CAREFULLY. ANY USE OF THIS TOOLKIT CONSTITUTES ACCEPTANCE OF THIS LICENSE.

IEMS API License Agreement

BY USING THE IEMS API MODULE OR THE IEMS API ITSELF, AS PART OR IN CONJUNCTION WITH, APPLICATIONS DEVELOPED, DISTRIBUTED OR IMPLEMENTED BY YOU OR ON YOUR BEHALF, YOU ARE CONSENTING TO BE BOUND BY THE TERMS AND CONDITIONS SET FORTH. IF YOU DO NOT AGREE WITH THESE TERMS, DO NOT USE THE IEMS API MODULE OR THE IEMS API TOOLKIT ITSELF.

GRANT OF LICENSE: IMA grants you a non-exclusive, non-transferable license to use the API Toolkit and accompanying documentation, as part or in conjunction with, applications developed, distributed or implemented by you or on your behalf. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code, with or without modification, must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form, with our without modification, must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

3. All advertising materials mentioning features or use of this software must display the following acknowledgment:

   "This product includes software developed by the International Messaging Associates Corporation for use with the Internet Exchange Messaging Server (IEMS). (http://www.ima.com)"

4. The names "International Messaging Associates", "IMA", "Internet Exchange Messaging Server", and "IEMS" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact sales@ima.com.

5. Redistributions of any form whatsoever must retain the following acknowledgment:

   "This product includes software developed by the International Messaging Associates Corporation for use with the Internet Exchange Messaging Server (IEMS). (http://www.ima.com)"

RIGHTS OF IMA: You acknowledge that title and any rights to the documentation and any copy made by you remain the sole and exclusive property of IMA. Any unauthorized modification and translation of the of the source code or documentation is strictly prohibited. Any breach or other failure to comply with the terms and conditions herein will entitle IMA to terminate this license and seek all other appropriate legal remedies.

LIMITATION OF LIABILITY: THIS SOFTWARE IS PROVIDED BY INTERNATIONAL MESSAGING ASSOCIATES CORPORATION "AS IS" AND ANY EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL INTERNATIONAL MESSAGING ASSOCIATES CORPORATION OR ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

GOVERNMENT RESTRICTED RIGHTS LEGEND (Applicable to U.S. Government End-Users Only)
Use, duplication or disclosure by the United States Government is subject to restrictions of Restricted Rights for computer software developed at private expense as set forth in FAR Sec. 52.227-19 or DOD FAR Supplement Sec. 252,227-7013(c)(1)(ii), and successor thereof, as applicable.

MISCELLANEOUS: This agreement will be governed and construed in accordance with the substantive laws of the State where delivery of the software occurred. If such delivery did not occur within a State or Territory of the United States, then this agreement shall be governed by the substantive laws of Hong Kong, and will in either case be without application of conflict or law principles.This agreement is the entire agreement and supersedes any other communications or advertising with respect to the software and accompanying documentation. Any modification of this agreement must be in writing and signed by an officer of IMA. If any provision of this agreement is held invalid, the remainder of this agreement will continue in full force and effect. *If you have any questions, please write us in this address:* **IMA Services Limited, 1303 Keen Hung Commercial Building, 80 Queen's Road East, Wan Chai, Hong Kong.**

INDEX

P

Q

S

W