Programmer's Manual

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

ISBN: **962-8137-38-7**
Document ID: **IEMS7PROGMA001**
Date of Publication: **June, 2003**

CONTENTS

PREFACE

The Internet Exchange Messaging Server (IEMS) 7 supports both a C++ Message Queue Application Programming Interface (MQAPI) and a C++ / PHP Client Application Programming Interface (IEMSCAPI). The MQAPI permits submission and retrieval of messages to and from the IEMS Message Queue (Message Transfer Agent), while the IEMSCAPI is used for the writing of messaging enabled applications, including web mail clients.

This manual is intended for C++ and PHP 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 and IEMSCAPIs 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	Detailed Discussion of IEMS Modules
Chapter 1	Message Queue API
Chapter 2	MQ API Class Definitions
Chapter 3	MQ API Function Reference
Chapter 4	How To Use The MQ API
Chapter 5	IEMS Client API
Chapter 6	Client API C++ Interface
Chapter 7	Client API PHP Interface
Appendix A	MQ API Test Program - testmq.c
Appendix B	MQ API Error Codes
Appendix C	Client API Constants
Appendix D	Message Store Folder Naming Issues
Appendix E	License Agreement

OVERVIEW

Background

The Internet Exchange Messaging Server (IEMS) is a highly modular and scalable open architecture messaging system that complies with Internet standards to ensure smooth and reliable transmission of messages. It can be used from small singel machine installations to fully distributed systems linking geographically distributed sites into a common set of logical domains. At the heart of IEMS is a number of components which work together to send and receive messages (see Figure 1 on page 7).

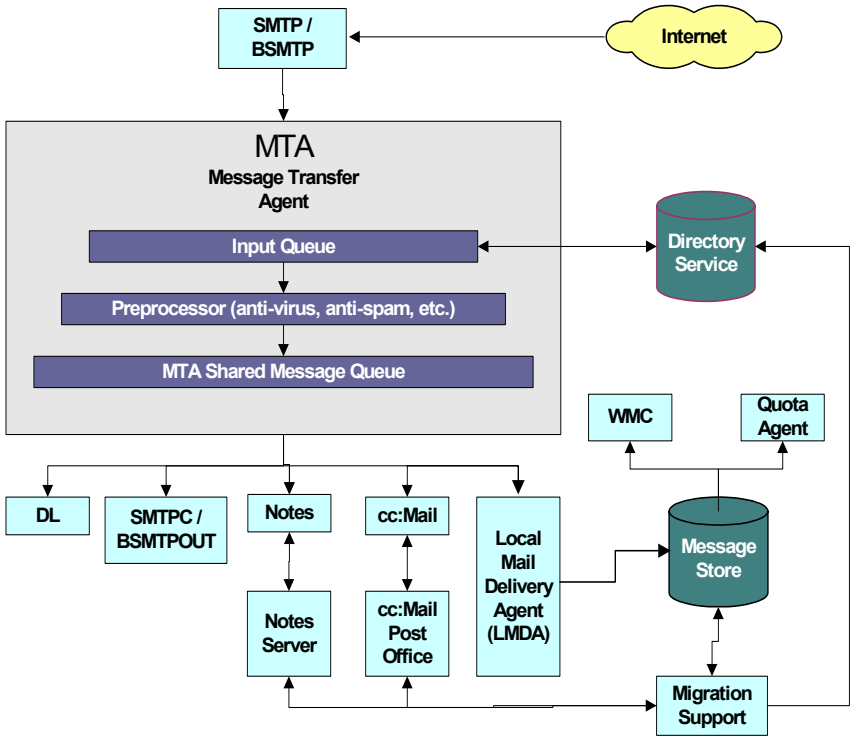

Figure 1: IEMS System Architecture

IEMS System Architecture

IEMS input channels receive messages from different applications supported by IEMS. These include SMTPD (SMTP traffic from the Internet / Intranet), Distribution List Manager, Web Mail Client, and others. IEMS 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.
- **CCOUT** - imports messages from the Lotus cc:Mail 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 MTA input queue. The Preprocessor takes these messages, and then performs address resolution/ expansion, virus scanning, spam checks, disclaimer insertion before inserting them into the MTA Shared Message Queue.

IEMS output channels take messages from the MTA Shared Message Queue and deliver them to specific applications supported by IEMS. These include SMTPC (outbound SMTP traffic to the Internet / Intranet), Distribution Lists, Local Message Store, and others. IEMS supports the following output channels:

- **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

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.

IEMS APIs

IMA provides two sets of Application Programming Interfaces (APIs) for messaging system developers. Developers looking to build gateway modules, or other applications that need to tightly integrate with the IEMS MTA and Preprocessor should use the Message Queue (MQ) API. This API provides the tools necessary to directly manipulate the MTA Shared Message Queue. In addition, programmers can make use of this API to build new Preprocessor filter modules.

Developers wanting to write user applications or other applications that sit outside of the messaging system should use the Client API's. The IEMS Client API provides both C++ as well as PHP interfaces to the application developer. It encapsulates most of the functional details provided by the different IEMS subsystems and provides a simplified API. The Client API provides a simple to use interface to the IEMS Message Store, and provides simple tools for message submission. User authentication and password management tools are also included.

Introduction

The Internet Exchange Messaging Server (IEMS) is a highly modular and scalable open architecture system. It can be used from small single machine installations to fully distributed systems linking geographically distributed sites into a common set of logical domains (see Figure 1 on page 7). Its various components can be run on a single machine or in a distributed environment.

IEMS 7 introduces a new integrated Anti-Spam approach to message reception and delivery. The MTA Pass-Through technology employed by IEMS 7 allows end users (message store accounts), individual distribution list maintainers, and connector modules to define their own security profiles independent of the rest of the system. At the same time the messaging system administrator can still define an overall global security policy, where some anti-spam measures will be handled directly by the MTA (such as reliable DNS-BL identified traffic). Other measures which may be desired by part of the user community, such as DNS-BL's with known high false positive rates (at the time of this writing, SpamCop and a few others have received a lot of industry coverage for their perceived indiscriminate listing practices) can then be passed through to the users for consultation on a case by case basis.

In most conventional messaging systems, security measures are employed on a system wide basis, making the choice of tools, such as DNS-BL's, critical. IEMS MTA Pass-Through technology changes this by allowing the administrator to be able to employ many more countermeasures, enabling only those that have been proven to be universally effective at the MTA level, and letting users pick and choose what additional measures they may or may not wish to apply to their individual message traffic.

IEMS Modules

Other IEMS modules include the MTA / Preprocessor, Directory Services, Distribution List Manager, SMTPC, SMTPD, BSMTP, Message Store, LMDA, and the MTA Shared Message Queue.

MTA / Preprocessor

The MTA is a message switch responsible for routing mail messages received by the Preprocessor to the intended channels. Upon receiving messages, the MTA temporarily stores the messages locally in a shared message queue while analyzing the recipient's address. It will either route the message to the recipient's local address or forward the mail to another MTA.

The Preprocessor Unit is an integrated subsystem of the MTA. It is equipped with anti-spam and anti-virus plug-in modules to protect the system against viruses and spam mail. It incorporates an auto text insertion engine, providing the capability to insert disclaimers into messages passing through the

MTA. Channel Action Matrices provide the system administrator with a flexible tool in configuring which plug-in modules should be run for a particular message based upon message flow through the system.

Directory Services

IEMS Directory Services are designed to effectively manage information about users, groups, mailing lists, alias processing and mail routing. It has a rich set of searching capabilities that makes directory lookup fast and efficient.

Distribution List Manager

The Distribution List Manager allows messages to be sent to list subscribers by simply submitting messages to a single address. It enables the system administrator to create electronic mailing lists that support the following features: mail blocking, automatic mailing list subscription and un-subscription, and setting the preferred delivery options. It also provides the system administrator with an option to accept or reject subscribers to the mailing list.

SMTPC (SMTP Client)

SMTPC delivers messages to the Internet. It provides fast mail delivery by processing messages based on their priority weight and by assigning different processors for deferred and pending messages.

SMTPD (SMTP Daemon)

SMTPD listens for incoming messages on the Internet. It is capable of sustaining simultaneous SMTP connections by creating multiple threads, thereby minimizing delay in message delivery.

BSMTP

Batch SMTP (BSMTP) tunnels messages so that they can pass through non-SMTP transports, such as POP3 (Post Office Protocol version 3). The original envelope and delivery information of each message is maintained across the tunnel.

Message Store

The Message Store acts as a dedicated mail repository for storing, retrieving and manipulating messages, while also enabling users to access their mailboxes via any POP3- and/or IMAP4-capable client. Users may also access their mail from the Message Store using the IEMS Web Mail Client, or any third-party application written around the Open Client API.

LMDA (Bayesian Filtering / MailSort)

Messages destined for a local user's Message Store account pass through the Local Mail Delivery Agent (LMDA) prior to delivery. The LMDA consists of the User Spam Controls, Bayesian Filter Engine, and the MailSort Engine (see Figure 2). Each of these three modules perform certain filtering and/or message filing operations on behalf of the user. Unlike similar operations that some mail clients use, these actions are performed by the messaging system, and at the time of message delivery. Once these modules are optionally configured by the user, their actions are transparent, as their mail client is not involved, and the actions happen as soon as messages arrive.

The User Spam Control module looks for messages that have been tagged by the MTA as potential spam for one or more reasons. This can be due to DNS-BL tagging (with the offending BL or BL's identified), and/or as a result of content filtering. Users can choose for each control if they want to act upon the tagging or not, and an appropriate action to take (ignore, discard, or file in the user specified spam folder).

The Bayesian Filtering module utilizes a statistical technique for spam detection based upon the users database of offending spam messages. Each user will have a different database based upon message they have individually categorized as spam according to their wishes. Messages caught by the Bayesian Filtering module can be either discarded, filed in a spam or suspicious folder, or passed to the MailSort engine for further processing.

The MailSort engine performs simple header pattern matching for the purpose of automatic filing of incoming messages as well as a vacation utility that is capable of sending vacation notifications during periods when the recipient is away.

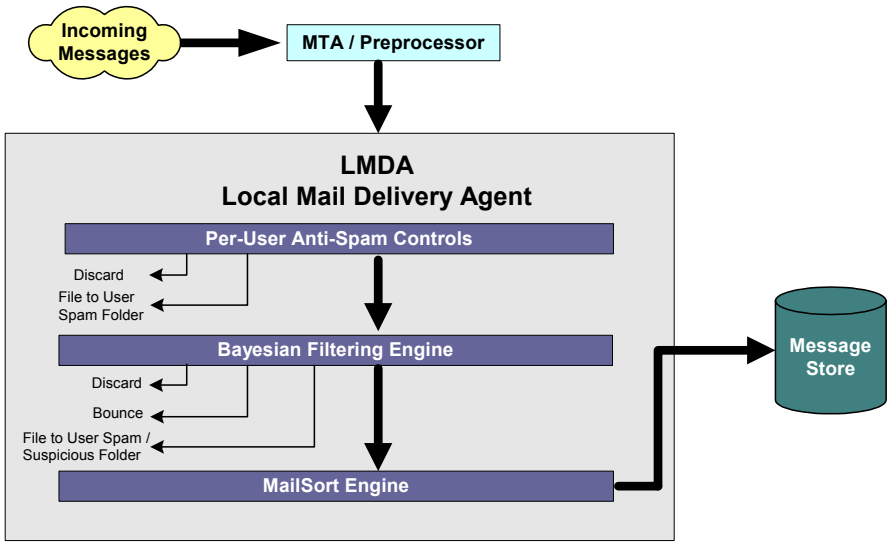

Figure 2:LMDA Architecture

MTA Shared Message Queue

Temporarily stores messages inserted by the Preprocessor Unit after preprocessing the messages for virus scanning, spam control, among others. Later they will be retrieved by the respective output channel processors for delivery to the intended recipients or downstream MTA's.

CHAPTER 1

Message Queue API

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 3 on page 15).

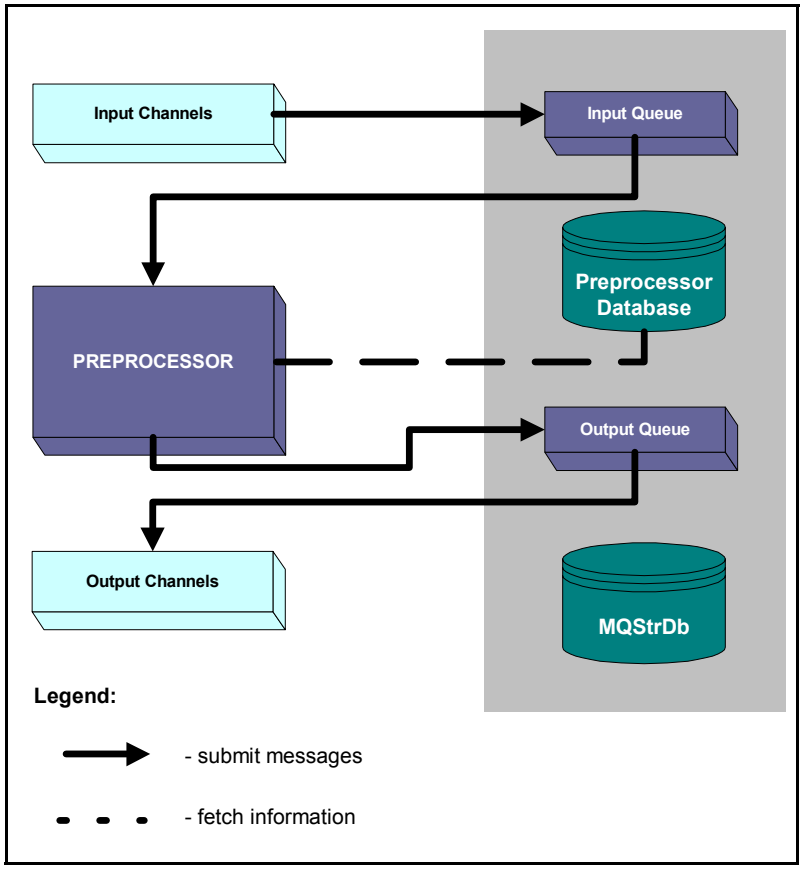

Figure 3: Message Flow

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 4 on page 16) and submits its messages to its respective input queue in the Message Queue Server.

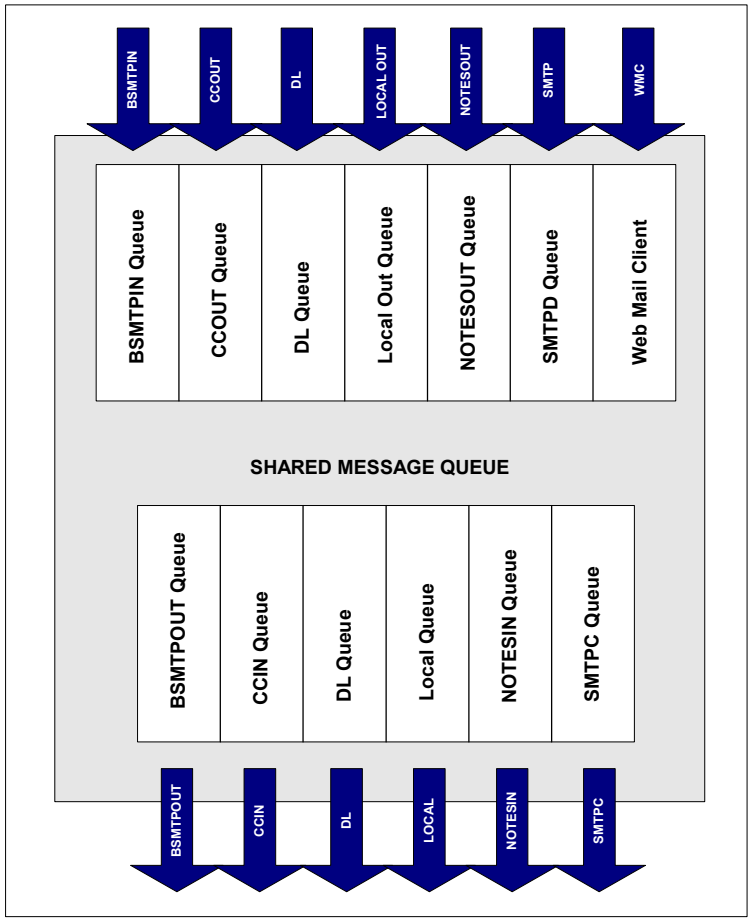

Figure 4: Channel I/O Mapping

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 5 on page 17), 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.

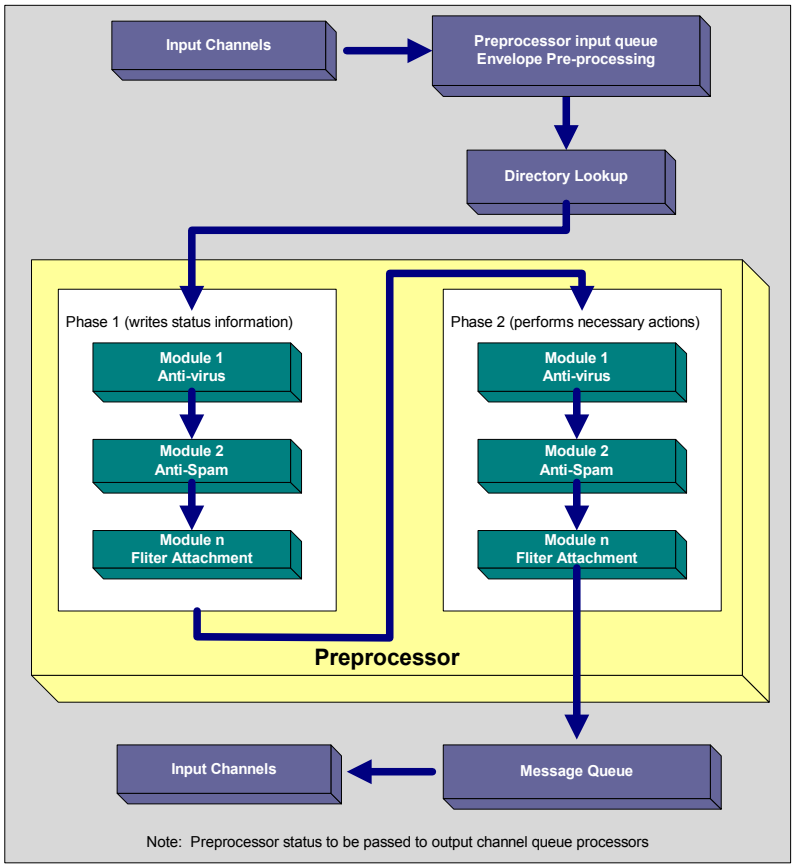

Figure 5: Tasks Performed by the Preprocessor

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 6 on page 19).

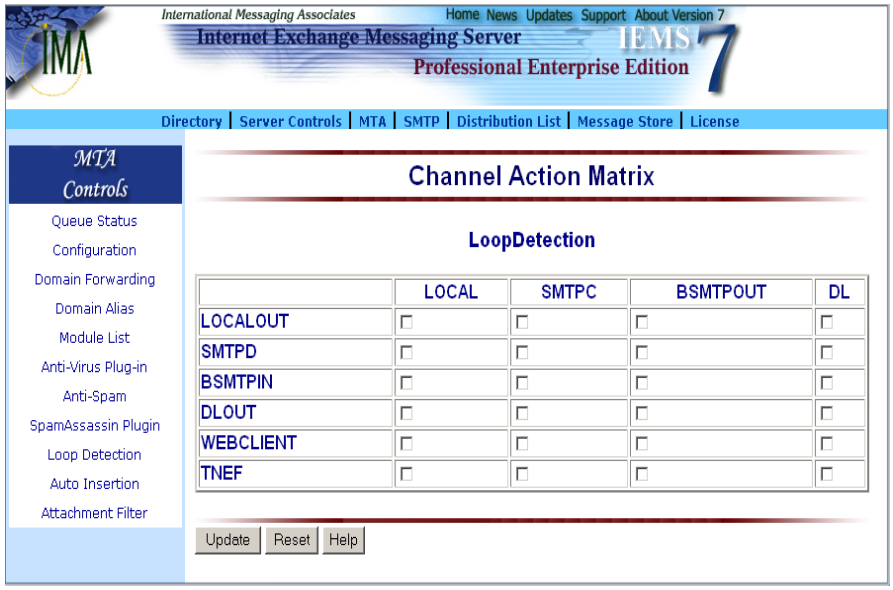

Figure 6: Channel Action Matrix

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 2

MQ 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 3

MQ 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 4

How To Use The MQ API

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 200 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 6.2 - 80, Caldera OpenLinux, Mandrake 8.2 - 9.1)
IEMS 7 for Linux
Compiler: gcc 2.91.66 or above

Software (Windows)

Windows NT/ 2000/XP
IEMS 7 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 7 on page 35).

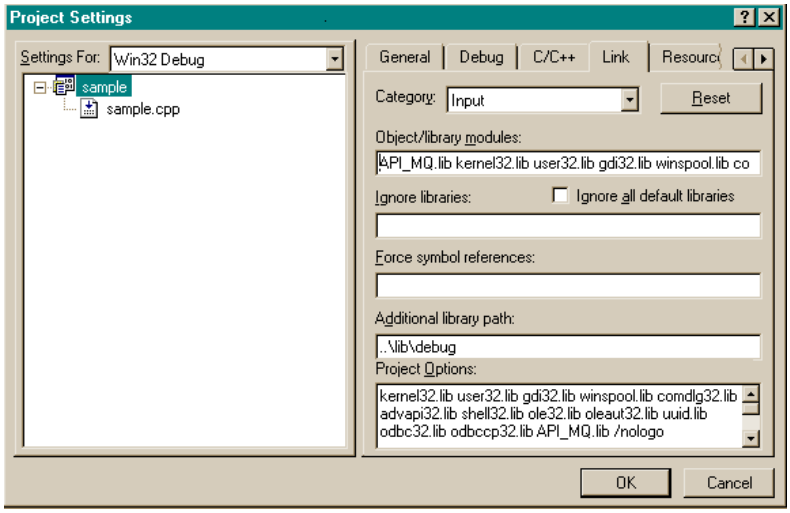

Figure 7: The Project Setting Dialog box

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);

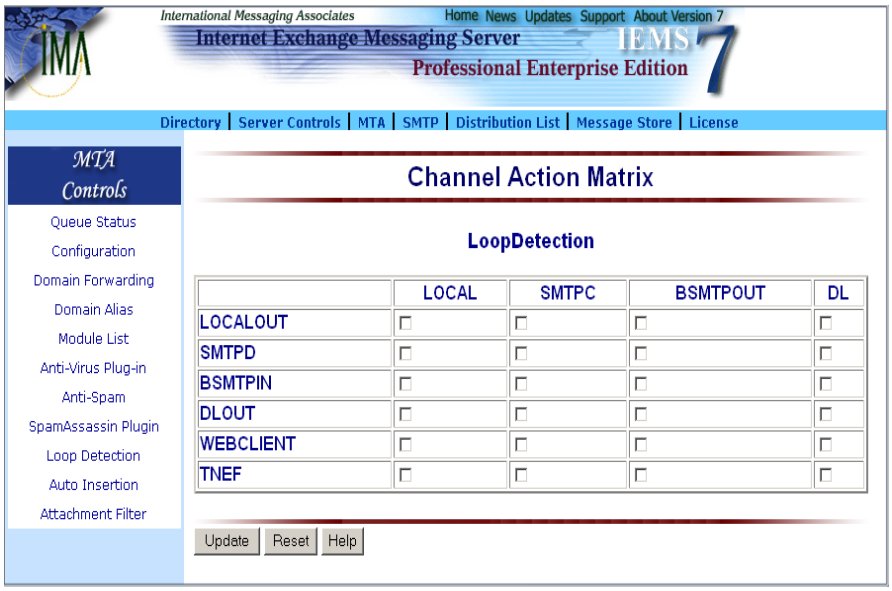

Figure 8: The Channel Action Matrix

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.

CHAPTER 5

IEMS Client API

This section describes the IEMS Client API and how it can be used as the basis for developing messaging enabled applications. It provides simple to use C++ and PHP interfaces on top of the various IEMS subsystems, including the Message Store, Directory Server, and MTA (see Figure 9 below). The Client API is a wrapper to the Message Store API, Message Queue API, and other added components (address book, signature, password, and session control).

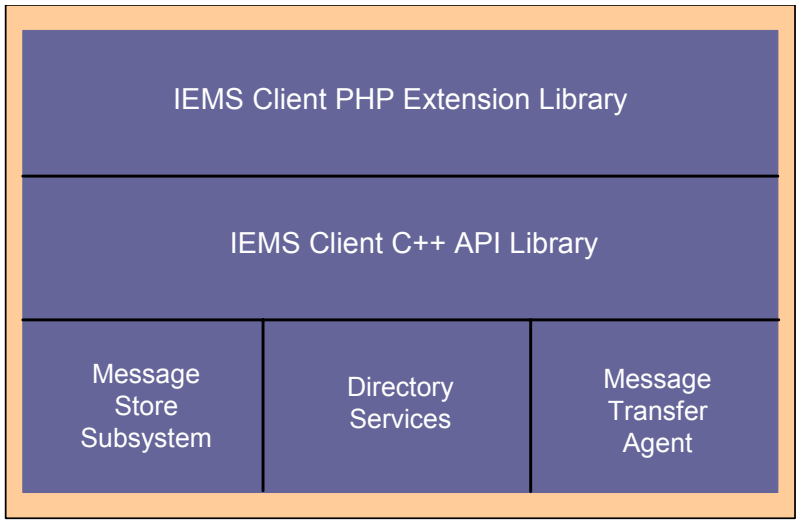

Figure 9: IEMS Client API

Tools provided within the Client API that cover the following functional areas:

- Authentication and Password Management
- Message Store Access
- Message Submission

Authentication / Password Management

Before any actions can be taken with respect to message store access, users first must identify themselves to the messaging system. This is done through loggin in to IEMS (Authenticate function). Additional functions are provided to logout from the system (Logout) and for changing user passwords (UpdatePassword).

Message Store Access

The Message Store Access routines provide a full set of functions in the following areas:

- Folder Access
- Message Access
- Message Header and Content Access

Folder Access

Folder Access routines allow the caller to create, rename, or delete folders for a given message store user. When folders names are made up of non-ASCII or double byte characters (e.g. Chinese, Japanese, and others), folder names are encoded in UTF-8 format. The Client API engine encodes the folder name to be modified UTF-7 format at stated in section 5.1.3 of RFC-2060. For Windows operating systems, the on-disk folder name is modified such that the hexadecimal portion of the UTF-7 encoded name is appended.

Folder access routines also provide methods for enumerating each folder and its attributes for a message store user. Folder Access routines include the following:

- CreateFolder
- RenameFolder
- DeleteFolder
- ReadFolderAttributes
- GetAllFoldernames
- FreeFoldernames

Message Access

The Message Access routines are used for accessing each message, its attributes, and its contents in the message store. It provides methods methods for the copying, moving, and deleting of messages from the message store, and methods for changing message attributes. Message Access routines include the following:

- GetUIDs
- GetUIDsWithSearchKey
- GetPrevNextUID
- GetPrevNextUIDWithSearchKey
- GetMessageInfo
- CopyMessage
- MoveMessage
- DeleteMessage
- MarkMessageAsRead

Message Header and Content Access

Messages received by IEMS are stored in their native Internet format, including any and all MIME structuring. The Message Header and Content Access routines provide the programmer with tools for the analysis and retrieval of such messages, and any attchments.

In addition to simple unstructured messages (non-MIME), there are 3 basic types of structured messages that are commonly encountered:

- Single Body MIME
- Multi-Part MIME
- Message / RFC-822

Single Body MIME

Single body MIME messages are the simplest structured messages. They are just text messages, with a character set identifier.

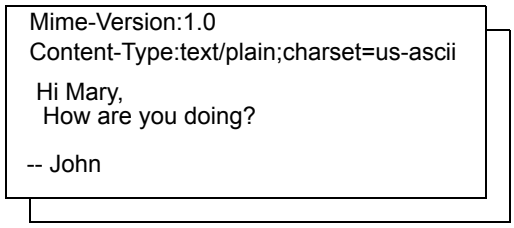

Figure 10: Single Body MIME

Multi-Part MIME

Multi-part MIME messages are used when trying to send multiple objects in a single message. For instance, in the example below (see Figure 11), a text message block and a image are included. The blocks are separated by the boundary string “*abc*”. Each block is essentially a separate MIME object. There can be an arbitrary number of attachments to a given message.

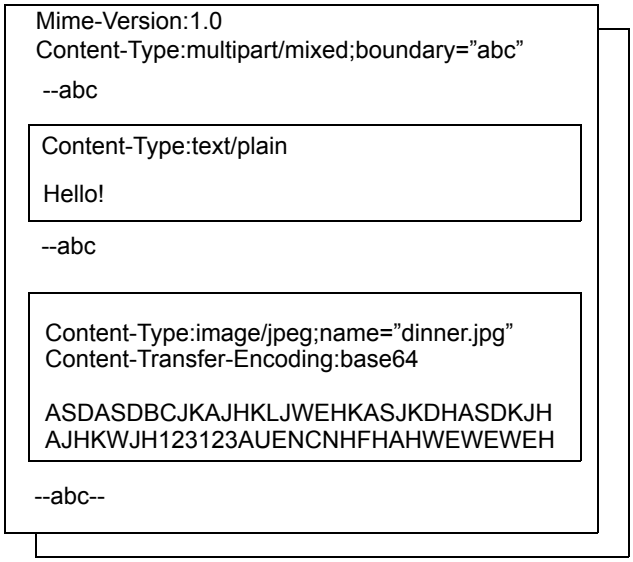

Figure 11: Multi-Part MIME

Message / RFC-822

The Message / RFC-822 content type is used when attaching or including another entire mail message inside a message. The primary MME headers simply identify the message as a MIME message, and that the contents are to be treated as a separate message. This format is typically used when forwarding messages with no added comments or content.

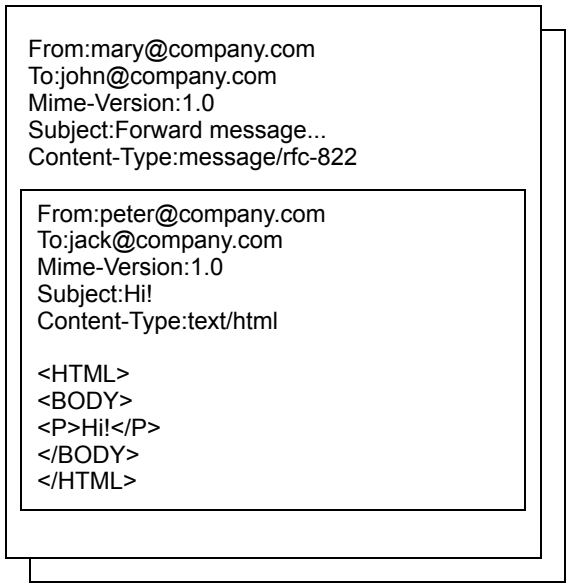

Figure 12: Message / RFC-822

Message Submission

ComposeMail is used to build and submit messages to the IEMS MTA. It can be used to build simple text only messages, or multipart MIME messages. While this routine is used for the composition and submission of messages to the MTA, it is important to note that the uploading of messages from a client workstation to the IEMS server is outside the scope of this function, and must be locally implimented for situations requiring file attachment.

Sample Applications

Web Mail Client

The IEMSCPHP toolkit includes a Web Mail Client application to demonstrates the usage of the IEMSCPHP extension library. This WMC application serves as a guideline on how to use various *iemsc_* functions to implement a web mail client interface for IEMS. You can add site banners, using different style sheet, adding images or so on to enrich the user interface. Or, if you do not like the layout at all, you can design your own web mail client interface by using all the available functions in the IEMSCPHP extension library.

login.htm
This provides the web mail login page to the user.

menu.htm
This provides the menu bar on the web mail client interface.

vfolder.htm
Provides a summary view of a folder with pagination support. You can copy, move or delete messages in a folder. This page uses:

- iemsc_authenticate
- iemsc_readfolderattributes
- iemsc_getuids
- iemsc_getmessagesinfo
- iemsc_utf7_to_decimal
- iemsc_isread
- iemsc_readallfoldernames

viewmsg.htm
Provides a view to display individual message content in a folder and provides links to reply or forward the message. It also provides navigation links to read the previous or next message available in a folder. Moreover, it demonstrates how to make use of the result from *iemsc_getmessagestructure* to display individual MIME body part on the page, or display a link to download a MIME body part as attachment. This page uses:

- iemsc_markmessageasread
- iemsc_getmessagestructure
- iemsc_getmessageinfo
- iemsc_getprevnextuid
- iemsc_getmessagebody
- iemsc_getembeddedheaders

viewextra.htm
Provides a view to display the full message headers and complete message source. This page uses:

- iemsc_getmessageheader
- iemsc_getmessagesource

getbody.htm
Provides a CGI interface to download an email attachment. This page uses:

- iemsc_getmessagebody

folder.htm
Provides a view to display all folders and their attributes. This pages uses:

- iemsc_renamefolder
- iemsc_deletefolder
- iemsc_createfolder
- iemsc_readallfoldernames
- iemsc_readfolderattributes_with_size
- iemsc_utf7_to_decimal
- iemsc_isspecialfolder

renfolder.htm
Provides a form to ask user to enter a new folder name for renaming. This page uses:

- iemsc_utf7_to_decimal

newmail.htm
Provides a form for a user to compose, reply and forward an email message and upload file attachments. This page uses:

- iemsc_getmessageinfo
- iemsc_getmessagestructure
- iemsc_getmessagesource
- iemsc_getmessagebody

sendmail.htm
This page submits the composing mail message to MQ. This page uses:

- iemsc_composemail

search.htm
Provides a message search form and displays the search result. This page uses:

- iemsc_readallfoldernames
- iemsc_copymessage
- iemsc_deletemessage
- iemsc_movemessage
- iemsc_searchuids
- iemsc_getmessagesinfo
- iemsc_utf7_to_decimal

- iemsc_isread

viewsearchmsg.htm
Provides a view to display the message content that matches the search criteria. This page uses:

- iemsc_markmessageasread
- iemsc_getmessagestructure
- iemsc_getmessageinfo
- iemsc_searchprevnextuid
- iemsc_getmessagebody
- iemsc_getembeddedheaders

passwd.htm
Provides a form to change the user password. This page uses:

- iemsc_updatepassword

wmc.php
This is the main PHP page that all CGI commands used by the Web Mail Client use. It uses *iemsc_authenticate* to verify the *username*, *hashedpassword* and *homedirectory* before passing the control to any of the above HTML pages. It demonstrates how to use $HTTP_SESSION_VARS to store the *'username'*, *'password'* and *'homedir'* information on the web server. This page uses:

- iemsc_authenticate
- iemsc_copymessage
- iemsc_deletemessage
- iemsc_movemessage
- iemsc_logout

CHAPTER 6

Client API C++ Interface

The IEMS Client C++ API is an open C++ API interface for 3rd party developers, used to write messaging applications on top of the IEMS messaging server. It encapsulates most of the functional details provided by the different IEMS subsystems and provides a simplified API interface for developers. The functions in the Client C++ API are specially designed for writing Web Mail Client and related applications for IEMS.

The IEMS Client C++ API provides functions that cover three major functional areas:

- Authentication and Password Management
- Message Store Access
- Message Submission

Descriptions of each of these functional areas can be found in Chapter 5.

Installation

The IEMS Client C++ API supports both Win32 and Linux operating systems. Microsoft Visual C++ 5.0 is used for Win32 operating systems and GCC 2.96 is used for Linux based systems. It is recommended to use the same tool set when compiling and linking your applications.

Before you can build IEMS enabled applications, you need to install and configure the IEMS Client API with your development environment. The IEMS API's can be found on your Version 7 installation CD, or can be downloaded from the IMA web site. For more details on how to download, see the IEMS API page at http://www.ima.com/iems/api.html. Once you have obtain the Client API distribution file, you can installed the Client C++ API library by following these procedures:

Microsoft Windows (Win32)

To install the Client API under Windows, perform the following steps:

1. Locate and unzip the iemsctoolkit.zip to your local harddisk. For example c:\iemsctoolkit

2. Copy c:\iemsctoolkit\lib\iemscapi.dll to your IEMS installation directory (c:\iems is the default location).

3. Make sure your IEMS installation directory is listed in the system PATH environment variable

4. Create a new project under VC++ 5.0 IDE

5. In your project source code, include the header file *#include "iemscapi.h"* and add c:\iemsctoolkit\include in your INCLUDE path

6. Add c:\iemsctoolkit\lib\iemscapi.lib to your project library path.

7. Make sure you are using the -MD flag in your project.

Linux

To install the Client API under Linux, perform the following steps:

1. Locate and untar the iemsctoolkit.tar file to your local harddisk. For example:

```
# mkdir /opt/iems/iemsctoolkit
# cd /opt/iems/iemsctoolkit
# tar xvf /tmp/iemsctoolkit.tar
```

2. Copy /opt/iems/iemsctoolkit/lib/libiemscapi.so to the IEMS library directory, ie. /opt/iems/lib

3. Run ldconfig -v to update your system library cache

4. In your project source code, include the header file #include "iemscapi.h" and add /opt/iems/iemsctoolkit/include in your INCLUDE path

5. Make sure you have defined -D_REENTRANT in your project makefile

6. Link your project with -L/opt/iems/lib -liemscapi -lpthread in your project makefile

7. Make sure you program is setuid / setgid to iems. Otherwise, your application will not have read/write access to IEMS storage area.

Software License

The IEMS Client C++ API library requires a proper IEMS message store license to operate. Please obtain the software license from www.ima.com and install the license properly before installing the API toolkit. Otherwise, the IEMS Client C++ library cannot operate properly.

The IEMSC Class

The IEMSC Class forms the foundation of the IEMSC C++ API. This class defines the methods used within the toolkit. Most methods return a IEMSC_ERR return code upon sucessful operation. All return codes are 32-bit unsigned long integers. The other return codes can be found in Appendix C.

Before any methods provided by the IEMSC class can be used, an iemsc object must be created. The *Authenticate* must be the first method called in order to create the proper user context.

This document only lists the exported public member functions of the IEMSC C++ class. The actual IEMSC class definition in the *iemscapi.h* file also includes the private methods and class emember variables that are not in the scope of this document.

The following defines the iemsc C++ class:

```
class iemsc
{
public:
  enum eSortKey {DEFAULT_KEY, BY_FROM, BY_TO, BY_DATE,
                 BY_SUBJECT, BY_SIZE, BY_UID};
  enum eSearchKey {NO_KEY, FROM_KEY, TO_KEY, DATE_KEY, SUBJECT_KEY};
  enum eSortDirection {DEFAULT_DIRECTION, ASCENDING, DESCENDING};
  IEMSC_DECL iemsc();
  IEMSC_DECL ~iemsc();


// Authentication methods
public:
  /*
   * The API application must call Authenticate method before
   * calling other API in the IEMSC class
   */
  IEMSC_DECL IEMSC_ERR Authenticate(const char *szUsername,
                            const char *szPassword,
                            const char *szCharset,
                            const char *szLocale,
                          const enum eSortDirection eSortDirection,
                            const enum eSortKey eSortKey,
                            char szHashPassword[],
                            bool bInit
                            );

  IEMSC_DECL IEMSC_ERR Authenticate(const char *szUsername,
                            const char *szPassword,
                            const char *szHomedir,
                            const char *szCharset,
                            const char *szLocale,
                          const enum eSortDirection eSortDirection,
                            const enum eSortKey eSortKey
                            );

  IEMSC_DECL IEMSC_ERR Logout(const char *szUsername,
                      const char *szHashPassword);


// Password management
  IEMSC_DECL IEMSC_ERR UpdatePassword(const char *szUsername,
                            const char *szOldpassword,
                            const char *szNewpassword);


// Folder management
  IEMSC_DECL IEMSC_ERR CreateFolder(const char *szFoldername);

  IEMSC_DECL IEMSC_ERR RenameFolder(const char *szOldFoldername,
                            const char *szNewFoldername);

  IEMSC_DECL IEMSC_ERR DeleteFolder(const char *szFoldername);

  IEMSC_DECL IEMSC_ERR ReadFolderAttributes(
```

```
                                const char *szFoldername,
                                unsigned long *ulNmsgs,
                                unsigned long *ulRecent,
                                unsigned long *ulUnseen,
                                unsigned long *ulUidnext,
                                unsigned long *ulSize
                                );

  IEMSC_DECL IEMSC_ERR GetAllFoldernames(char ***pFolderNames);

  IEMSC_DECL void FreeFoldernames(char **pFolderNames);


// Message access
  IEMSC_DECL IEMSC_ERR GetUIDs(const char *szFolderName,
                       const enum eSortDirection eSortDirection,
                       const enum eSortKey eSortKey,
                       unsigned long ** pUIDs,
                       unsigned long pagesize=0,
                       unsigned long pagenumber=0);

  IEMSC_DECL IEMSC_ERR GetUIDsWithSearchKey(
                       const char *szFolderName,
                       const enum eSortDirection eSortDirection,
                       const enum eSortKey eSortKey,
                       const enum eSearchKey eSearchKey,
                       const char *szSearchValue,
                       const time_t tBefore,
                       const time_t tAfter,
                       unsigned long ** pUIDs,
                       unsigned long *ulHit,
                       unsigned long pagesize=0,
                       unsigned long pagenumber=0);

  IEMSC_DECL IEMSC_ERR GetPrevNextUID(const char *szFolderName,
                       const enum eSortDirection eSortDirection,
                       const enum eSortKey eSortKey,
                       unsigned long ulUid,
                       unsigned long ulPagesize,
                       unsigned long *ulPrevUid,
                       unsigned long *ulNextUid,
                       unsigned long *ulPagenumber);

  IEMSC_DECL IEMSC_ERR GetPrevNextUIDWithSearchKey(
                       const char *szFolderName,
                       const enum eSortDirection eSortDirection,
                       const enum eSortKey eSortKey,
                       const enum eSearchKey eSearchKey,
                       const char *szSearchValue,
                       const time_t tBefore,
                       const time_t tAfter,
                       unsigned long ulUid,
                       unsigned long ulPagesize,
                       unsigned long *ulPrevUid,
                       unsigned long *ulNextUid,
                       unsigned long *ulPagenumber);
  IEMSC_DECL IEMSC_ERR GetMessageInfo(const char *szFolderName,
                       unsigned long ulUid,
                       char **pFrom,
                       char **pTo,
                       char **pCc,
                       char **pSubject,
                       char **pDate,
                       unsigned long &ulSize,
                       unsigned long &ulFlags);
```

```
  IEMSC_DECL IEMSC_ERR CopyMessage(const char *szSourceFolder,
                       const unsigned long ulUid,
                       const char *szDestinationFolder);

  IEMSC_DECL IEMSC_ERR MoveMessage(const char *szSourceFolder,
                       const unsigned long ulUid,
                       const char *szDestinationFolder);

  IEMSC_DECL IEMSC_ERR DeleteMessage(const char *szFolderName,
                       const unsigned long ulUid);

  IEMSC_DECL IEMSC_ERR MarkMessageAsRead(const char *szFolderName,
                       const unsigned long ulUid);

  IEMSC_DECL IEMSC_ERR GetMimeStructure(const char *szFolderName,
                       const unsigned long ulUid,
                       struct MIMEBODY ** pBody);

  IEMSC_DECL IEMSC_ERR GetMimeBody(const char *szFolderName,
                       unsigned long ulUid,
                       int iPartNumber,
                       char **pDecodeStream,
                       unsigned long &ulLength);

  IEMSC_DECL IEMSC_ERR GetMessageHeader(const char *szFolderName,
                       unsigned long ulUid,
                       char **pHeaderStream,
                       unsigned long &ulHeaderSize);

  IEMSC_DECL IEMSC_ERR GetMessageSource(const char *szFolderName,
                       unsigned long ulUid,
                       char **pMessageSource,
                       unsigned long &ulMessageSize);

  IEMSC_DECL IEMSC_ERR GetEmbeddedHeaders(const char *szFolderName,
                       unsigned long ulUid,
                       int iPartNumber,
                       char **pFrom,
                       char **pTo,
                       char **pCc,
                       char **pSubject,
                       char **pDate);


// Message creation ( submit to MQ )
  IEMSC_DECL IEMSC_ERR ComposeMail(const char *szCharset,
                       const char *szToAddress,
                       const char *szCcAddress,
                       const char *szBccAddress,
                       const char *szSubject,
                       const char *szMailbody,
                       const bool bHTMLbody,
                       const struct attachment *attach,
                       const int  nAttachment,
                       const bool bSaveToDraft,
                       const bool bSaveToOutbox);


// Supporting functions
  IEMSC_DECL void FreeString (char *pString);

  IEMSC_DECL void FreeBuffer (void *pBuffer);

  IEMSC_DECL void FreeMimeBody(struct MIMEBODY *mimebody);
```

```
    IEMSC_DECL bool IsSpecialFolder(char *szFoldername);

    IEMSC_DECL char *utf7_decimal(char *utf7);

    IEMSC_DECL char *GetHomeDirectory(void) { return m_szHomedir; };
};
```

Authentication / Password Management

Authenticate (Form 1)

Syntax: IEMSC_DECL IEMSC_ERR Authenticate

Parameters:

const char *szUsername,
const char *szPassword,
const char *szCharset,
const char *szLocale,
const enum eSortDirection eSortDirection,
const enum eSortKey eSortKey,
char szHashPassword[],
bool bInit

Returns: IEMSC_NO_ERR on success.

Description:
This is the first method to call before calling other methods in the IEMSC class (with the exception of *UpdatePassword* method). When *bInit* is set to true, you need to pass the clear text password in the *szPassword* field. The API will compute a per-session-hashed password and return it in the *szHashPassword* field. The *szHashPassword* size must be at least *HASH_PASSWORD_LENGTH* large (see iemscpai.h). When *bInit* is set to false, the per-session-hashed password needs to be passed in the *szPassword* field and the *szHashPassword* field is ignored. The function verifies the user name and password and initializes the user credential in the IEMS system. If the user name and / or password is not correct, all succeeding call to other methods in IEMSC class will always return *IEMSC_NOT_AUTHENTICATED*.

The *szCharset* and *szLocale* fields are used when submitting message into the MQ subsystem. If you specify an EMPTY STRING, the system default will be used.

The *eSortDirection* and *eSortKey* controls how the IEMS UIDs are being returned by the *GetUIDs* and *GetUIDsWithSearchKey* methods. You can specify *DEFAULT_DIRECTION* and *DEFAULT_KEY* respectively to use the system defaults, which are set to *DESCENDING* and *BY_UID*.

Note: *For a CGI type application, you can use bInit=true to first verify the clear text password entered by the user. Then the per-session-hashed password can be used to re-authenticate the user CGI session and prevent the CGI from passing the clear text password around.*

Authenticate (Form 2)

Syntax: IEMSC_DECL IEMSC_ERR Authenticate

Parameters:
const char *szUsername,
const char *szPassword,
const char *szHomedir,
const char *szCharset,
const char *szLocale,
const enum eSortDirection eSortDirection,
const enum eSortKey eSortKey

Returns: IEMSC_NO_ERR on success.

Description:
This is a variant of the above Authenicate method with *bInit=false*. The first Authenicate method always performs a LDAP query to the server to retrieve the location of the user's HOME directory. To achieve better performance, your application can pass the user HOME directoy in the *szHomedir* field to re-authenticate the user name and password.

Logout

Syntax: IEMSC_DECL IEMSC_ERR Logout

Parameters:
const char *szUsername,
const char *szHashPassword

Returns: IEMSC_NO_ERR on success.

Description:
Call this method to remove the user credentials created by the *Authenticate* method. If the function succeeds, the per-session-hashed password is invalidated. Therefore, you need to call the *Authenticate* method with *bInit=true* to get a new per-session-hashed password.

UpdatePassword

Syntax: IEMSC_DECL IEMSC_ERR UpdatePassword

Parameters:
const char *szUsername,
const char *szOldpassword,
const char *szNewpassword

Returns: IEMSC_NO_ERR on success.

Description:
Use this method to change the password for the given username. Both the *szOldpassword* and *szNewpassword* feilds are clear text. The new password must contain no less than 6 characters.

Message Folder Access

CreateFolder

Syntax: IEMSC_DECL IEMSC_ERR CreateFolder

Parameter:
const char *szFoldername

Returns: IEMSC_NO_ERR on success.

Description:
Use this method to create a new folder in the user's HOME directory. If your folder name contains non-ASCII characters say Chinese GB/BIG5, the *szFoldername* field must be in UTF8 encoding. See Appendix D for details.

RenameFolder

Syntax: IEMSC_DECL IEMS_ERR RenameFolder

Parameters:
const char *szOldFoldername,
const char *szNewFoldername

Returns: IEMSC_NO_ERR on success.

Description:
Use this method to rename an existing folder in the user's HOME directory. If your new folder name contains non-ASCII characters say Chinese GB/BIG5, the *szNewFoldername* field must be in UTF8 encoding. See Appendix D for details.

Note: *There are four special folders which cannot be renamed. They are the "inbox", "outbox", "trash" and "drafts" folders. If you attempt to rename one of these special folders, IEMSC_SPECIAL_FOLDER is returned.*

DeleteFolder

Syntax: IEMSC_DECL IEMSC_ERR DeleteFolder

Parameter:
const char *szFoldername

Returns: IEMSC_NO_ERR on success.

Description:
Use this method to delete an existing folder in the users HOME directory. If this folder still contains messages, the folder cannot be deleted and *IEMSC_FOLDER_NOT_EMPTY* will be returned.

Note: *There are four special folders cannot be deleted. They are the "inbox", "outbox", "trash" and "drafts" folders. If you attempt to rename one of these special folders, IEMSC_SPECIAL_FOLDER is returned.*

ReadFolderAttributes

Syntax: IEMSC_DECL IEMSC_ERR ReadFolderAttributes

Parameters:
const char *szFoldername,
unsigned long *ulNmsgs,
unsigned long *ulRecent,
unsigned long *ulUnseen,
unsigned long *ulUidnext,
unsigned long *ulSize

Returns: IEMSC_NO_ERR on success.

Description:
Use this method to read the attributes of a folder. There are 5 attributes associated with each folder. They are:

ulNMsgs - total number of messages
ulRecent - number of new messages added to this folder since last access
ulUnseen - number of unread messages in this folder
ulUidnext - the next available UID in this folder
ulSize - size of this folder

Note: *The size of the folder is computed every time this method is called. ulSize can be set to equal NULL if the application is not interested in the folder's size to achieve faster response.*

GetAllFoldernames

Syntax: IEMSC_DECL IEMSC_ERR GetAllFoldernames

Parameter:
char ***pFolderNames

Returns: IEMSC_NO_ERR on success.

Description:
This method is used to read all the available folders in the users HOME directory. If the folder name contains non-ASCII characters, the folder name is encoded in modified UTF-7 encoding (See Appendix D for details). The application must call FreeFoldernames method to release the buffer allocated by this method after used.

Example:

```
iemsc iemsc;
IEMSC_ERR err;
char **pFolders;
err = iemsc.Authenticate(...);
err = iemsc.GetAllFoldernames(&pFolders);
if (err == IEMSC_NO_ERR) {
char **p;
for (p = pFolders; p && *p != NULL; p++) {
printf("Folder name: %s\n", *p);
}
```

```
iemsc.FreeFolderNames(pFolders);
}
```

FreeFoldernames

Syntax: IEMSC_DECL void FreeFoldernames

Parameter:
char **pFolderNames

Returns: NONE

Description:
Use this method to release the buffer allocated by the *GetAllFoldernames* method. Failure to release this buffer after use will result in memory leakage.

Message UID Access

GetUIDs

Syntax: IEMSC_DECL IEMS_ERR GetUIDs

Parameters:
const char *szFolderName,
const enum eSortDirection eSortDirection,
const enum eSortKey eSortKey,
unsigned long ** pUIDs,
unsigned long pagesize=0,
unsigned long pagenumber=0;

Returns: IEMSC_NO_ERR on success

Description:
This method is used to read all the Message UIDs stored in a given folder. The Message UIDs are returned in an array that is sorted based on the value specified in the *eSortDirection* and *eSortKey* parameters. The last element in the returned array is always *ZERO*. If a non-zero pagesize is specified, only '*pagesize*' number of UIDs in the given *pagenumber* are returned. Therefore, your application can make use of the *pagesize* and *pagenumber* value to access chuck of UIDs at a time. Your application must call the *FreeBuffer* method to release the UIDs buffer allocated by this method after use.

Note: *If DEFAULT_DIRECTION and / or DEFAULT_KEY is specified in the eSortDirection and / or eSortKey fields, the values specified in the Authenticate method are used.*

Example:

```
iemsc iemsc;
IEMSC_ERR err;
unsigned long *uids, *puid;

err = iemsc.Authenticate(....);
```

```
/* only get 10 UIDs in page ZERO */
err = iemsc.GetUIDs("inbox", DEFAULT_DIRECTION, DEFAULT_KEY,
                    &uids, 10, 0);
if (err == IEMSC_NO_ERR) {
   puid = uids;
   while(*puid != 0) {
      printf("UID: %ld\n", *puid);
      puid++;
   }
iemsc.FreeBuffer(uids);
```

GetUIDsWithSearchKey

Syntax: IEMSC_DECL IEMSC_ERR GetUIDsWithSearchKey

Parameters:
const char *szFolderName,
const enum eSortDirection eSortDirection,
const enum eSortKey eSortKey,
const enum eSearchKey eSearchKey,
const char *szSearchValue,
const time_t tBefore,
const time_t tAfter,
unsigned long ** pUIDs,
unsigned long *ulHit,
unsigned long pagesize=0,
unsigned long pagenumber=0

Returns: IEMSC_NO_ERR on success.

Description:
The iemsc class provides simple search capability to find Message UIDs by matching keywords in message FROM, TO, SUBJECT fields, or by matching a date/date range.

When the *eSearchKey* field is set to *FROM_KEY*, *TO_KEY,* or *SUBJECT_KEY*, keyword searching is specified in the *szSearchValue* field. If *szSearchValue* is set to *NULL*, this method searches UIDs with the given *FROM*, *TO* or *SUBJECT* field absent in the message.

When *eSearchKey* field is set to *DATE_KEY*, this method searches UIDs that matches the time value defined in *tBefore* and / or *tAfter* fields. You can specify either tBefore or tAfter to (time_t)-1 if not searching message UID in a given date range.

The number of message that match the searching criteria is returned in the *ulHit* field. Similar to *GetUIDs* method, the *pagesize* and *pagenumber* fields can be used to retrieve chunks of UIDs at a time. The *FreeBuffer* method needs to be called in order to release the UIDs buffer allocated by this method after use.

Note: *If DEFAULT_DIRECTION and / or DEFAULT_KEY is specified in the eSortDirection and / or eSortKey fields, the values specified in the Authenticate method are used.*

Example:

```
/* Find message with subject line contains 'Hello' */
iemsc iemsc;
IEMSC_ERR err;
unsigned long *uids, *puid;
unsigned long nFound = 0;

err = iemsc.Authenicate(...);
err = iemsc.GetUIDsWithSeachKey("inbox",
                     DEFAULT_DIRECTORY,
                     DEFAULT_KEY,
                     SUBJECT_KEY,
                     "Hello",
                     -1,
                     -1,
                     &uids,
                     &nFound,
                     10,
                     0);

if (err == IEMSC_NO_ERR) {
   printf("Found %ld messages\n", nFound);
   puid = uids;
   while(*puid != 0) {
      printf("UID: %ld\n", *puid);
      puid++;
   }
   iemsc.FreeBuffer(uids);
```

GetPrevNextUID

Syntax: IEMSC_DECL IEMSC_ERR GetPrevNextUID

Parameters:

const char *szFolderName,
const enum eSortDirection eSortDirection,
const enum eSortKey eSortKey,
unsigned long ulUid,
unsigned long ulPagesize,
unsigned long *ulPrevUid,
unsigned long *ulNextUid,
unsigned long *ulPagenumber

Returns: IEMSC_NO_ERR on success.

Description:
This method is used to locate the previous and next message UID of a given message UID using a specified sort direction and sort key. If the *ulPagesize* field is not *ZERO*, this method also computes the page number that the given UID is located. If there are no previous and / or next UIDs in this sorted tree, the value of *ulPrevUid* and / or *ulNextUid* is set to *ZERO*.

Note: *If DEFAULT_DIRECTION and / or DEFAULT_KEY is specified in the eSortDirection and / or eSortKey fields, the values specified in the Authenticate method are used.*

Example:

```
iemsc iemsc;
IEMSC_ERR err;
unsigned long prev_uid, next_uid;
unsigned long pagernumber;

err = iemsc.Authenicate(...);
err = iemsc.GetPrevNextUid("inbox",
                DEFAULT_DIRECTION,
                DEFAULT_KEY,
                100,
                10,
                &prev_uid,
                &next_uid,
                &pagenumber);
if (err == IEMSC_NO_ERR) {
   if (prev_uid != 0)
      printf("Previous uid: %ld\n", prev_uid);
   if (next_uid != 0)
      printf("Next uid: %ld\n", next_uid);
   printf("UID 100 is located in page: %ld\n", pagenumber);
```

GetPrevNextUIDWithSearchKey

Syntax: IEMSC_DECL IEMSC_ERR GetPrevNextUIDWithSearchKey

Parameters:
const char *szFolderName,
const enum eSortDirection eSortDirection,
const enum eSortKey eSortKey,
const enum eSearchKey eSearchKey,
const char *szSearchValue,
const time_t tBefore,
const time_t tAfter,
unsigned long ulUid,
unsigned long ulPagesize,
unsigned long *ulPrevUid,
unsigned long *ulNextUid,
unsigned long *ulPagenumber

Returns: IEMSC_NO_ERR on success.

Description:
GetPrevNextUIDWithSearchKey is similar to the *GetPrevNextUID* method but with simple searching capability. See the *GetUIDsWithSearchKey* method on the how to search message in a folder.

Note: *If DEFAULT_DIRECTION and / or DEFAULT_KEY is specified in the eSortDirection and / or eSortKey fields, the values specified in Authenticate method are used.*

Example:

```
iemsc iemsc;
IEMSC_ERR err;
unsigned long prev_uid, next_uid;
unsigned long pagernumber;
time_t now;
time_t tBefore;

/* we look for message received two days ago */
now = time(NULL);
tBefore = now - ( 24 * 60 * 60 * 2);
err = iemsc.Authenicate(...);
err = iemsc.GetPrevNextUidWithSearchKey("inbox",
                 DEFAULT_DIRECTION,
                 DEFAULT_KEY,
                 DATE_KEY,
                 NULL,
                 tBefore,
                 (time_t)-1,
                 100,
                 10,
                 &prev_uid,
                 &next_uid,
                 &pagenumber);
if (err == IEMSC_NO_ERR) {
   if (prev_uid != 0)
      printf("Previous uid: %ld\n", prev_uid);
   if (next_uid != 0)
      printf("Next uid: %ld\n", next_uid);
   printf("UID 100 is located in page: %ld\n", pagenumber);
```

GetMessageInfo

Syntax: IEMSC_DECL IEMSC_ERR GetMessageInfo

Parameters:

const char *szFolderName,
unsigned long ulUid,
char **pFrom,
char **pTo,
char **pCc,
char **pSubject,
char **pDate,
unsigned long &ulSize,
unsigned long &ulFlags

Returns: IEMSC_NO_ERR on success.

Description:
This method is used to get the message *FROM*, *TO*, *CC*, *DATE* and *SUBJECT* header fields as well as the message size and flags. The *TO* and *CC* fields can contain one or more email addresses which are separated by a *COMMA* in the return buffer. The application needs to call the *FreeBuffer* method to release the buffers returned in the *pFrom*, *pTo*, *pCc*, *pSubject* and *pDate* fields.

The *ulFlags* fields contain a bitwise ORed message flag. Applications can use the *IS_XXX* macro defined in *IEMSCAPI.H* to test if a certain message flag is ON or OFF.

CopyMessage

Syntax: IEMSC_DECL IEMSC_ERR CopyMessage

Parameters:
const char *szSourceFolder,
const unsigned long ulUid,
const char *szDestinationFolder

Returns: IEMSC_NO_ERR on success.

Description:
Use this method to copy a message from one folder to another.

MoveMessage

Syntax: IEMSC_DECL IEMSC_ERR MoveMessage

Parameters:
const char *szSourceFolder,
const unsigned long ulUid,
const char *szDestinationFolder

Returns: IEMSC_NO_ERR on success.

Description:
Use this method to move a message from one folder to another.

DeleteMessage

Syntax: IEMSC_DECL IEMSC_ERR DeleteMessage

Parameters:
const char *szFolderName,
const unsigned long ulUid

Returns: IEMSC_NO_ERR on success.

Description:
Use this method to delete a message from a folder.

MarkMessageAsRead

Syntax: IEMSC_DECL IEMSC_ERR MarkMessageAsRead

Parameters:
const char *szFolderName
const unsigned long ulUid

Returns: IEMSC_NO_ERR on success.

Description:
Use this method to set the *SEEN* bit in the message flag. It will also update the '*unseen*' attribute in the given folder.

Message Header / Content Access

GetMimeStructure

Syntax: IEMSC_DECL IEMSC_ERR GetMimeStructure

Parameters:
const char *szFolderName,
const unsigned long ulUid,
struct MIMEBODY ** pBody

Returns: IEMSC_NO_ERR on success.

Description:
This method is used to get the MIME tree structure for a given message UID. The struct MIMEBODY is defined as follows:

```
struct MIMEBODY
{
  char *ct;   /* content-type */
  char *cst;  /* content sub type */
  char *cte;  /* content transfer encoding */
  char *name; /* name parameter in content-type */
  char *filename; /* filename parameter in content-disposition */
  char *charset; /* charset parameter in cotent-type */
  char *ctdisp;  /* content disposition */
  unsigned long offset;
  unsigned long size;
  struct MIMEBODY **child;
  unsigned int n_child;
  int part_number;  /* identifer of a body part in the
                       MIME structure */
};
```

When accessing a single MIME body message, this method returns an array with a single element only. The '*part_number*' of this MIME body is set to 0, and the *child* pointer is NULL, and *n_child* is zero.

For Multipart MIME messages (see Figure 11 on page 42), this method returns a struct MIMEBODY 'm' with *n_child* equal to 2, the child pointer set, and the *part_number* of this outmost MULTIPART/MIXED MIME entity is 0. The child pointer can be accessed via indexed array (index 0 and 1) which stores the MIME structure of the TEXT/PLAIN and IMAGE/JPEG MIME entity respectively. The part_number of the TEXT/PLAIN and IMAGE/JPEG MIME entity is 1 and 2 respectively

For Message / RFC-822 messages (see Figure 12 on page 42), this method returns a struct MIMEBODY 'm' with *n_child* equal to 1, the child pointer is set, and the *part_number* of this outmost MESSAGE/RFC822 MIME entity

set to 0. The child pointer can be accessed via indexed array (index 0) which stores the MIME structure of the TEXT/HTML MIME entity. The *part_number* of the TEXT/HTML MIME entity is 1.

The application must call the *FreeMimeBody* method to release the struct MIMEBODY pointer after use.

Example:

```
void traverse(struct MIMEBODY *m)
{
   printf("Content-type: %s/%s\n", m->ct, m->cst);
   if (m->n_child > 0) {
      unsigned long i;
      for (i = 0; i < m->n_child; i++) {
         traverse(m->child[i]);
         }
      }
   }

int main(int argc, char *argv[]) {
/* traverse each MIME entity in the tree */
iemsc iemsc;
IEMSC_ERR err;
struct MIMEBODY *m=NULL;

err = iemsc.Authenticate(...);
err = iemsc.GetMimeStructure("inbox", 100, &m);
if (err == IEMSC_NO_ERR) {
   traverse(m);
   iemsc.FreeMimeBody(m);
   }
}
```

GetMimeBody

Syntax: IEMSC_DECL IEMSC_ERR GetMimeBody

Parameters:
const char *szFolderName,
unsigned long ulUid,
int iPartNumber,
char **pDecodeStream,
unsigned long &ulLength

Returns: IEMSC_NO_ERR on success.

Description:
This method is used to get the decoded byte stream of a MIME entity in a message. *iPartNumber* is used to address which body part is to be decoded. The decoded byte stream is returned in the *pDecodeStream* pointer and the length of the decoded byte stream is returned in the *ulLength* field. The application needs to call the *FreeBuffer* method with the *pDecodeStream* after use.

Example:

```
iemsc iemsc;
IEMSC_ERR err;
char *decode_data = NULL;
unsigned long len;
err = iemsc.Authenticate(...);
err = iemsc.GetMimeBody("inbox", 1, 0, &decode_data, len);
if (err == IEMSC_NO_ERR) {
   printf("Length of decoded byte stream: %ld\n", len);
   iemsc.FreeBuffer(decode_data);
   }
```

GetMessageHeader

Syntax: IEMSC_DECL IEMSC_ERR GetMessageHeader

Parameters:
const char *szFolderName,
unsigned long ulUid,
char **pHeaderStream,
unsigned long &ulHeaderSize

Returns: IEMSC_NO_ERR on success.

Description:
This method is used to get the full RFC822 message header for a given message. The header is returned in the *pHeaderStream* pointer and its length is returned in the *ulHeaderSize* field.

GetMessageSource

Syntax: IEMSC_DECL IEMSC_ERR GetMessageSource

Parameters:
const char *szFolderName,
unsigned long ulUid,
char **pMessageSource,
unsigned long &ulMessageSize

Returns: IEMSC_NO_ERR on success.

Description:
This method is used to get the complete message content for a given message. The message content is returned in the *pMessageSource* pointer and its length is returned in the *ulMessageSize* field.

Note: *In the IEMS MessageStore, each message is stored as a separate file under the user's HOME directory. Each message is a RFC822 / MIME formatted message with a 6-byte binary header at the beginning of the message. If your application tries to interpret the message content by itself, it should skip the first 6 octets of the message.*

GetEmbeddedHeaders

Syntax: IEMSC_DECL IEMSC_ERR GetEmbeddedHeaders

Parameters:
const char *szFolderName,
unsigned long ulUid,
int iPartNumber,
char **pFrom,
char **pTo,
char **pCc,
char **pSubject,
char **pDate

Returns: IEMSC_NO_ERR on success.

Description:
This function is used to read the 5 header fields of an embedded RFC822 MIME entity. In the example in Figure 12, we have two MIME entities. The first one is MESSAGE/RFC822 with *iPartNumber* equal to 0. The second one is TEXT/HTML with *iPartNumber* equal to 1. We can use this method to read the embedded message header (i.e. *From: peter@company.com* ..) by setting *iPartNumber* equal to 0. The application needs to call the *FreeString* method with the *pFrom*, *pTo*, *pCc*, *pSubject* and *pDate* fields to release the allocated memory after use.

NOTE: *The embedded headers always belong to the MESSAGE/RFC822 MIME entity with iPartNumber = 0 in this sample message*.

Message Submission

ComposeMail

Syntax: IEMSC_DECL IEMSC_ERR ComposeMail

Parameters:
const char *szCharset,
const char *szToAddress,
const char *szCcAddress,
const char *szBccAddress,
const char *szSubject,
const char *szMailbody,
const bool bHTMLbody,
const struct attachment *attach,
const int nAttachment,
const bool bSaveToDraft,
const bool bSaveToOutbox

Returns: IEMSC_NO_ERR on success.

Description:
This method is used to submit a message into the MQ subsystem (Message Transfer Agent). The *szCharset* field controls the charset parameter to be used in the generated message. An EMPTY STRING can be specified such

that the charset value specified in *Authenticate* method is used. The *szToAddress*, *szCcAddress* and *szBccAddress* fields specify the Internet email address in the message *TO*, *CC* and *BCC* headers respectively. These are the recipient addresses for message delivery. If there are more than a single address in any of these fields, use a COMMA to separate each of them. At least one recipient address must be specified in *szToAddress*, *szCcAddress* or *szBccAddress*. If not, IEMSC_ERR_NO_RECIPIENT is returned. The message subject is set in the *szSubject* field. The *szMailbody* field contains the message content. If *szMailbody* is HTML formatted, set *bHTMLbody* to true, otherwise, set it to false. One or more attachments in the message can be specified in the attach field. The attachment structure is defined as:

```
struct attachment
{
  char *on_disk_file_name;
  char *display_name;
  char *ct;
  char *cst;
  char *cte;
};
```

The *on_disk_file_name* field sets the location of the attachment on the local filesystem. The *display_name* field suggests the name to be displayed by the recipient email agent.

Note: *In the Linux version, your application must have READ permission for the file specified in on_disk_file_name.*

The *ct* and *cst* fields suggest the content type and content subtype of this file attachment and the *cte* field suggests the encoding method for this attachment. This can be base64, quoted-printable, 7bit or 8bit. If you set *ct*, *cst* and *cte* to NULL, the system will try to lookup the mapping for you based on the file extension in the display_name field. If no mapping can be found, the default Content-Type will be set to APPLICATION/OCTET-STREAM and the encoding set to BASE64. If there is no attachment in the message, set *nAttachment* to ZERO. If the *bSaveToDraft* field is set to TRUE, the message is saved to the user's 'drafts' folder but not submitted to MQ subsystem. If *bSaveToOutbox* field is set to true, a copy of the message is saved to the user's 'outbox' folder after being submitted to MQ subsystem. When *bSaveToDraft* is set, the value of *bSaveToOutbox* field is ignored.

Example:

```
iemsc iemsc;
IEMSC_ERR err;
struct attachment att;
att.on_disk_file_name="c:\\temp\\temp123.dat"
att.display_name="my note.txt"
att.ct="text"
att.cst="basic"
att.cte="7bit"
err = iemsc.Authenticate(...);
err = iemsc.ComposeMail("US-ASCII",
                  "john@company.com,mary@company.com",
                  "",
                  "",
                  "Test message",
                "Hi, this is a test message.\r\nPlease ignore\r\n",
```

```
                                    false,
                                    &att,
                                    1,
                                    false,
                                    true);
```

Other Functions

FreeString

Syntax: IEMSC_DECL void FreeString

Parameter:
char *pString

Returns: NONE

Description:
Releases the string buffer allocated by various iemsc methods.

FreeBuffer

Syntax: IEMSC_DECL void FreeBuffer

Parameter:
void *pBuffer

Returns: NONE

Description:
Releases the buffer allocated by various iemsc methods.

FreeMimeBody

Syntax: IEMSC_DECL void FreeMimeBody

Parameter:
struct MIMEBODY *mimebody

Returns: NONE

Description:
Releases the struct MIMEBODY buffer allocated by the *GetMimeStructure* method.

IsSpecialFolder

Syntax: IEMSC_DECL bool IsSpecialFolder

Parameter:
char *szFoldername

Returns: true or false

Description:
Tests if the given folder name is a *SPECIAL* folder. A *SPECIAL* folder cannot be renamed or deleted. The 4 special folders are *inbox*, *outbox*, *trash* and *drafts*.

utf7_decimal

Syntax: IEMSC_DECL char *utf7_decimal

Parameter:
char *utf7

Returns: A string pointer to a &#DDDDD; encoded stream

Description:
In the IEMS MessageStore, when a folder contains non-ASCII characters, the folder name is encoded in a modified UTF7 encoding scheme as stated in RFC2060. This modified UTF7 encoding is not supported by many browsers. Applications can use this method to convert UTF7 encoded folder names to &#DDDDD; representation which is supported by most browsers. The FreeString method needs to be used to release the return buffer after use. See Appendix D for details on UTF7 encoding.

GetHomeDirectory

Syntax: IEMSC_DECL char *GetHomeDirectory

Parameters:
NONE

Returns: A string pointer to the location of the users HOME directory

Description:
This method is used to retrieve the users HOME directory. Applications can save this value and call the second form of the *Authenticate* method to re-authenticate the user.

CHAPTER 7

Client API PHP Interface

The IEMS Client PHP API is an open PHP API interface for 3rd party developers, used to write messaging applications on top of the IEMS messaging server. It encapsulates most of the functional details provided by the different IEMS subsystems and provides a simplified API interface for developers. The functions in the Client PHP API are specially designed for writing Web Mail Client and related applications for IEMS. As PHP is a very common and simple web programming language to use, web designer and web application programmers can easily modify web mail client interfaces included in the IEMSCPHP API toolkit to fulfill their specific requirements.

The IEMS Client PHP API provides functions that cover three major functional areas:

- Authentication and Password Management
- Message Store Access
- Message Submission

Descriptions of each of these functional areas can be found in Chapter 5.

Installation

The IEMSCPHP extension library is complied against PHP version 4.3.0. The revision numbers are listed below:

PHP API	20020918
PHP Extension	20020429
Zend Extension	20021010

If you are not using this version of PHP in your system, the IEMSCPHP extension library cannot be used in your server. Please obtain the PHP source code from *www.php.net*. Alternatively, for Win32 operating systems, you can download the pre-built binary from *www.php.net*. Please consult the PHP home page on how to install and configure PHP with your Apache server. Once PHP 4.3.0 is properly installed and configured with your Apache server, you can continue to install the IEMSCPHP extension on your system.

Before you can build IEMS enabled applications, you need to install and configure the IEMS Client API with your development environment. The IEMS API's can be found on your Version 7 installation CD, or can be downloaded from the IMA web site. For more details on how to download, see the IEMS API page at http://www.ima.com/iems/api.html. Once you have obtain the Client API distribution file, you can installed the Client PHP API library by following these procedures:

Microsoft Windows (Win32)

To install the Client API under Windows, perform the following steps:

1. Locate and unzip the iemsctoolkit.zip to your local harddisk. For example c:\iemsctoolkit

2. Copy c:\iemsctoolkit\lib\iemscapi.dll to your IEMS installation directory (c:\iems is the default location).

3. Copy c:\iemsctoolkit\lib\iemscphp.dll to your PHP extension library directory (for example, c:\php-4.3.0-win32\lib).

4. Copy c:\iemsctoolkit\iemsc*.* to the IEMS htdocs directory, for example: c:\iems\apache\htdocs\iems\iemsc

5. Add *extension=iemscphp.dll* to the PHP.INI.file.

6. Make sure your IEMS installation directory is listed in the system PATH environment variable

7. Restart the Apache server

Linux

To install the Client API under Linux, perform the following steps:

1. Locate and untar the iemsctoolkit.tar file to your local harddisk. For example:

```
# mkdir /opt/iems/iemsctoolkit
# cd /opt/iems/iemsctoolkit
# tar xvf /tmp/iemsctoolkit.tar
```

2. Copy /opt/iems/iemsctoolkit/lib/libiemscapi.so to the IEMS library directory, ie. /opt/iems/lib

3. Copy /opt/iems/iemsctoolkit/lib/iemscphp.so to your PHP extension library directory (i.e. /usr/lib/php4)

4. Copy /opt/iems/iemsctoolkit/iemsc/*.* to the IEMS htdocs directory (for example /opt/iems/htdocs/iems/iemsc).

5. Add *extension=iemscphp.so* to the PHP.INI file.

6. Run ldconfig-v to update your system library cache.

7. Modify your apache httpd.conf file so that Apache is started with UID=iems and GID=iems.

Note: *Running Apache as the IEMS user is necessary for the PHP libraries to operate properly. If you need to run several virtual servers running with different UID's, see http://httpd.apache.org/docs/suexec.html for instructions.*

8. Restart the Apache server

Note: *Please remove the extension=ldap.so in your PHP.INI file as IEMS uses its own version of libldap32.so that is not compatible with the ldap.so distributed by PHP.*

Once the system is ready, you can start using the sample applications included in the IEMSCPHP toolkit. Start your favorite browser and connect to

http://<hostname>/iems/iemsc/login.htm

To try out the PHP based Web Mail Client.

Software License

The IEMS Client PHP API library requires a proper IEMS message store license to operate. Please obtain the software license from www.ima.com and install the license properly before installing the API toolkit. Otherwise, the IEMS Client PHP library cannot operate properly.

Authentication / Password Management

iemsc_authenticate

Parameters:
String username,
String password,
String locale,
String charset,
long sort_direction,
long sort_key,
bool init

Returns: This function return an array where:
array[0] contains the error code

When init flag is set to true;
array[1] contains the per session hashed password
array[2] contains the location of the user HOME DIRECTORY

Description:
This function is used with init=*true* to verify the username (ie. user@domain) and the clear text password. If the username and password are correct, IEMSC_NO_ERR is returned in array[0]. The IEMSC C++ API subsystem will compute a per session hashed password and return the value in array[1]. The location of the user HOME DIRECTORY is returned in array[2]. The application should employ some logic to store this hashed password and home directory that are used by other functions in IEMSCPHP extension library. The locale and charset parameters can be set to EMPTY STRING. In this case, the system-wide locale and charset value will be used. The *sort_direction* and *sort_key* values control the sorting order of UIDs. IEMSC_DEFAULT_SORT_DIRECTION can be set to use the system wide

default. The same holds true for IEMSC_SORT_BY_DEFAULT_KEY. See Appendix C for details.

When init flag is *false*, the application should pass the per session hashed password to the password field for checking. In this case, the application should employ its own logic to store this per session hashed password for succeeding *iemsc_authenticate* calls. Applications should always call *iemsc_authenticate* with init=*false* to update the last access time stamp before calling other functions in the IEMSCPHP extension library. If the session has expired, IEMSC_SESSION_EXPIRED is returned in array[0]

iemsc_logout

Parameters:
String username,
String hashedpassword

Returns: IEMSC_NO_ERR on success.

Description:
This function is used to clear the login session information of the given username.

iemsc_updatepassword

Parameters:
String username,
String oldpassword,
String newpassword

Returns: IEMSC_NO_ERR on success.

Description:
This function is used to change the password of the given username. The caller must pass the correct *oldpassword* for verification. Both *oldpassword* and *newpassword* must be in cleartext. *newpassword* must contain at least 6 characters.

Message Folder Access

iemsc_createfolder

Parameters:
String username,
String hashedpassword,
String homedir,
String newfoldername

Returns: IEMSC_NO_ERR on success.

Description:
This function is used to create a new folder for the given username. The value of *hashedpassword* and *homedir* are used to verify the provided user-

name. If *newfoldername* contains non-ASCII characters, a UTF-8 encoded string should be used (See Appendix D for details)

iemsc_renamefolder

Parameters:
String username,
String hashedpassword,
String homedir,
String oldfoldername,
String newfoldername

Returns: IEMSC_NO_ERR on success.

Description:
This function is used to rename an existing folder to *newfoldername*. The value of *hashedpassword* and *homedir* are used to verify the provided username. If *newfoldername* contains non-ASCII characters, a UTF-8 encoded string should be used (See Appendix D for details).

iemsc_deletefolder

Parameters:
String username,
String hashedpassword,
String homedir,
String foldername,

Return value: IEMSC_NO_ERR on success.

Description:
This function is used to delete an existing folder. The value of *hashedpassword* and *homedir* are used to verify the provided username. If the specified folder is not empty, IEMSC_FOLDER_NOT_EMPTY is returned and the folder is not deleted.

iemsc_readfolderattributes

Parameters:
String username,
String hashedpassword,
String homedir,
String foldername,
Long Reference nmsg,
Long Reference recent,
Long Reference unseen,
Long Reference uidnext

Returns: IEMSC_NO_ERR on success.

Description:
This function is used to read the 4 attributes, *nmsg* (number of message),

recent (number of recent message since last access to this folder), *unseen* (number of unread message) and *uidnext* (the next available UID of this folder) of the specified folder. The value of *hashedpassword* and *homedir* are used to verify the provided username. The caller should pass a reference to the variables for storing *nmsg*, *recent*, *unseen* and *uidnext*.

Example:

```
$nmsg = 0;
$recent = 0;
$unseen = 0;
$uidnext = 0;
$ret=iemsc_readfolderattributes("user@company.com",
         "A456718183803",
         "c:\iems\msgstore\user@company.com",
         "inbox",
         &$nmsg,
         &$recent,
         &$unseen,
         &$uidnext);
```

iemsc_readfolderattributes_with_size

Parameters:

String username,
String hashedpassword,
String homedir,
String foldername,
Long Reference nmsg,
Long Reference recent,
Long Reference unseen,
Long Reference uidnext,
Long Reference size

Returns: IEMSC_NO_ERR on success.

Description:
This function is used to read the 5 attributes, *nmsg* (number of message), *recent* (number of recent message since last access to this folder), *unseen* (number of unread message), *uidnext* (the next available UID of the folder) and *size* of the specified folder. The value of *hashedpassword* and *homedir* are used to verify the provided username. The caller should pass a reference to the variables for storing *nmsg*, *recent*, *unseen*, *uidnext* and *size*.

Note: *This function needs to perform additional computation to get the size of the specified folder name. For performance reason, use this function only if you need to know the size of a folder. Use iemsc_readfolderattributes instead when the size attribute is not needed.*

iemsc_readallfoldernames

Parameters:
String username,
String hashedpassword,
String homedir

Returns: array containing individual folder names in each array element on success, otherwise normal error code.

Description:
This function is used to read all the available folders of the given username. The value of *hashedpassword* and *homedir* are used to verify the provided username. Folder name needs to be encoded in modified UTF7 format if it contains non-ASCII characters (See Appendix D for details).

Example:

```
$ret=iemsc_readallfoldernames("user@company.com",
         "A456718183803",
         "c:\iems\msgstore\user@company.com");

/* make sure it is not an error code */
if (is_array($ret)) {
   for ($i = 0; $i < count($ret); $i++) {
      echo "Folder $i -> $ret[$i]\n";
      }
   }
```

Message UID Access

iemsc_getuids

Parameters:
String username,
String hashedpassword,
String homedir,
String foldername,
Long sort_direction,
Long sort_key,
Long pagesize,
Long pagenumber

Returns: an array of Long integers containing individual UIDs on success. The UIDs are sorted based on the sort_direction and sort_key value specified in this function. Otherwise, a normal error code is returned.

Description:
This function is used to read available message UIDs of the specified foldername of the user. The value of *hashedpassword* and *homedir* are used to verify the given username. When this function succeeds, an array of UIDs is returned. The UIDs are sorted based on the *sort_key* and *sort_direction* flag specified in this function (See Appendix C for details). When *pagesize* is set to zero, all available UIDs are returned. Otherwise, only *pagesize* number of UIDs are returned for the given *pagenumber*. Therefore, an application can make use of the *pagesize* and *pagenumber* value to implement a pagination view for a folder.

Example:

```
/*
 * read the first 10 uids of inbox sorted by size in
 * ascending order
*/
$ret=iemsc_readallfoldernames("user@company.com",
                "A456718183803",
                "c:\iems\msgstore\user@company.com",
                "inbox",
                IEMSC_SORT_BY_SIZE,
                IEMSC_SORT_DIRECTION_ASCENDING,
                10,
                0);
if (is_array($ret)) {
   for ($i = 0; $i < count($ret); $i++) {
      echo "Uid: $ret[$i]\n";
      }
   }
```

iemsc_getprevnextuid

Parameters:

String username,
String hashedpassword,
String homedir,
Sting foldername,
Long uid,
Long pagesize

Returns: On success, an associative array with following keys is returned:

```
hash['prev']   /* the previous uid before the given uid */
hash['next']   /* the next uid after the given uid */
hash['pagenumber'] /* the current page number of the given
                      uid with the given pagesize */
```

On failure - normal IEMSC error code

Description:

This function is used to locate the previous and next uid and the current page-number of the given UID under the given foldername. The value of *hashed-password* and *homedir* are used to verify the given username. This function uses the value of *sortkey* and *sortdirection* stored in the login session for sorting UIDs. If *prev*/*next* uid equals to ZERO, there is no previous or next uid in the sorted tree.

iemsc_searchuids

Parameters:

String username,
String hashedpassword,
String homedir,
String foldername,
Long sort_direction,
Long sort_key,
Long search_key,
String searchvalue,

Long time_before,
Long time_after,
Long pagesize,
Long pagenumber,
Long Reference matchcount

Returns: On success, an array of UIDs that match the searching criteria is returned, otherwise normal IEMSC error code.

Description:
This is a function similar to *iemsc_getuids* but with simple searching capability. Applications can search UIDs based on keyword matching in *From*, *To*, or *Subject* fields, or search UIDs based on a date or a date range (See Appendix D for details). The value of *hashedpassword* and *homedir* are used to verify the given username.

When IEMSC_SEARCH_BY_FROM, IEMSC_SEARCH_BY_TO or IEMSC_SERACH_BY_SUBJECT search_key are used, EMPTY STRING can be specified as searchvalue to search message with the corresponding field absent in the mail message.

When using IEMSC_SEARCH_BY_DATE search_key, you need to supply *time_before* and/or *time_after* value. These values are in time_t format. Use -1 in either *time_before* or *time_after* if you are not searching message in a given time range.

Similar to the *iemsc_getuids* function, a non-zero *pagesize* and a *pagenumber* can be used for implementing paginated views. The matchcount returns total number of messages that match the searching criteria.

iemsc_searchprevnextuid

Parameters:
String username,
String hashedpassword,
String homedir,
String foldername,
Long sort_direction,
Long sort_key,
Long search_key,
String searchvalue,
Long time_before,
Long time_after,
Long pagesize

Returns: On success, an associative array with following keys is returned:

```
hash['prev']     /* the previous uid before the given uid */
hash['next']     /* the next uid after the given uid */
hash['pagenumber'] /* the current page number of the given uid
                      with the given pagesize */
```

On failure - normal IEMSC error code.

Description:
This is a function similar to *iemsc_getprevnextuid* but with simple searching capability. Applications can search previous and next UIDs of the given uid based on keyword matching in *From*, *To*, or *Subject* fields, or search UIDs based on a date or a date range (See Appendix C for details). The value of *hashedpassword* and *homedir* are used to verify the given username. If *prev*/*next* uid equals to ZERO, there are no previous or next uids in the sorted search tree.

iemsc_getmessageinfo

Parameters:

String username,
String hashedpassword,
String homedir,
String foldername,
Long uid

Returns: On success, an associative array with the following keys:

```
hash['from']     /* a String contains the From header */
hash['to']       /* a String contains all the addresses in the To header,
                    each address is separated by COMMA */
hash['cc']       /* a String contains all the addresses in the Cc header,
                    each address is separated by COMMA */
hash['subject']  /* a String contains the subject header */
hash['date']     /* a String contains the date header */
hash['size']     /* an Long integer of the message size */
hash['flag']     /* an Long integer of the message system flag */
```

On failure - normal IEMSC error code.

Description:
This function is used to read the 5 header fields (*from*, *to*, *cc*, *subject* and *date*) and the *size* and *message* flag of the given uid in a folder. The value of *hashedpassword* and *homedir* are used to verify the given username. The *To* and *Cc* header may contain more than one addresses. A COMMA is used to separate each address in the To and Cc header field.

iemsc_getmessagesinfo

Parameters:

String username,
String hashedpassword,
String homedir,
String foldername,
Array uids

Return values: On success, an array of associative arrays with the following keys are returned:

```
hash['from']     /* a String contains the From header */
hash['to']       /* a String contains all the addresses in the To header,
                    each address is separated by COMMA */
hash['cc']       /* a String contains all the addresses in the Cc header,
                    each address is separated by COMMA */
```

hash['subject'] /* a String contains the subject header */
hash['date'] /* a String contains the date header */
hash['size'] /* an Long integer of the message size */
hash['flag'] /* an Long integer of the message system flag */
On failure, normal IEMSC error code returned.

Description:
This function is used to read the 5 header fields (*from*, *to*, *cc*, *subject* and *date*) and the *size* and *message* flag of the given uids array in a folder. The value of *hashedpassword* and *homedir* are used to verify the given username. The *To* and *Cc* header may contain more than one addresses. A COMMA is used to separate each address in the *To* and *Cc* header fields. Use this function instead of *iemsc_getmessageinfo* if you want to read more than one UID message info at a time.

Example:

```
$uids=array(10, 11, 12);
$ret=iemsc_getmessagesinfo("user@company.com",
              "A456718183803",
              "c:\iems\msgstore\user@company.com",
              "inbox",
              $uids);
if (is_array($ret)) {
   for ($i = 0; $i < count($ret); $i++) {
      echo "Subject for UID $uids[$i] is $ret[$i]['subject']\n";
      }
   }
```

iemsc_copymessage

Parameters:
String username,
String hashedpassword,
String homedir,
String srcfoldername,
Long uid
String destfoldername

Returns: IEMSC_NO_ERR on success.

Description:
This function is used to copy a message from a source folder to a destination folder. The value of *hashedpassword* and *homedir* are used to verify the given username.

iemsc_movemessage

Parameters:
String username,
String hashedpassword,
String homedir,
String srcfoldername,
Long uid
String destfoldername

Returns: IEMSC_NO_ERR on success.

Description:
This function is used to move a message from source folder to destination folder. The value of *hashedpassword* and *homedir* are used to verify the given username.

iemsc_deletemessage

Parameters:
String username,
String hashedpassword,
String homedir,
String foldername,
Long uid

Returns: IEMSC_NO_ERR on success.

Description:
This function is used to delete a message from a folder. The value of *hashed-password* and *homedir* are used to verify the given username.

iemsc_markmessageasread

Parameters:
String username,
String hashedpassword,
String homedir,
String foldername,
Long uid

Returns: IEMSC_NO_ERR on success.

Description:
This function is used to set the SEEN flag of a message in a folder. The value of *hashedpassword* and *homedir* are used to verify the given username.

Message Header / Content Access

iemsc_getmessagestructure

Parameters:
String username,
String hashedpassword,
String homedir,
String foldername,
Long uid

Returns: On success, an array of associative arrays with the following keys:

```
hash['ct']          /* String, content type */
hash['cst']         /* String, content subtype */
hash['cte']         /* String, content transfer encoding */
hash['name']        /* String, 'name' parameter in content-type header */
```

```
hash['filename']      /* String, 'filename' parameter in
                         content-disposition header */
hash['ctdisp']        /* String, content disposition header */
hash['charset']       /* String, charset parameter in content-type header */
hash['partnumber']    /* Long, the 'partnumber' of a body part */
hash['num_child']     /* Long, number of CHILD body part of this
                         Multipart/* or Message/RFC822 entity */
On failure, normal IEMSC_ERR code
```

Description:
This function is used to read the structure of a MIME formatted message of a given uid in a folder. The value of *hashedpassword* and *homedir* are used to verify the given username.

When accessing a single MIME body message, the *iemsc_getmessagestructure* function returns an array with 1 element only. The *partnumber* of this MIME body is 0 and *num_child* of this MIME body is also 0.

When accessing a Multipart MIME message (see Figure 11 on page 42), the *iemsc_getmessagestructure* function returns an array with 3 elements. Array[0] contains the MIME structure of the outermost MULTIPART/MIXED MIME entity with *partnumber* set to 0 and *num_child* set to 2. Array[1] contains the MIME structure of the TEXT/PLAIN MIME entity with *partnumber* set to 1. Array[2] contains the MIME structure of the IMAGE/JPEG MIME entity with *partnumber* equal to 2.

When accessing a Message / RFC-822 MIME message (see Figure 12 on page 42), the *iemsc_getmessagestructure* function returns and array with 2 elements. Array[0] contains the MIME structure of the outermost MESSAGE/RFC822 MIME entity with *partnumber* set to 0 and *num_child* set to 1. Array[1] contains the MIME structure of the TEXT/HTML MIME entity with *partnumber* equal to 1.

iemsc_getmessagebody

Parameters:

```
String username,
String hashedpassword,
String homedir,
String foldername,
Long uid,
Long partnumber
```

Returns: On success, a string that contains the decoded data of the MIME entity with the given partnumber, otherwise normal IEMSC_ERR code.

Description:
This function is used to read the decoded data stream of given *uid* and *partnumber*. The value of *hashedpassword* and *homedir* are used to verify the given username. This function decodes the MIME entity based on the encoding method specified in the content-transfer-encoding header in that MIME entity.

iemsc_getmessageheader

Parameters:

String username,
String hashedpassword,
String homedir,
String foldername,
Long uid

Returns: On success, a string containing the full RFC822 message headers for the given uid, otherwise a normal IEMSC_ERR code.

Description:
This function is used to read the full RFC822 message headers for a given uid. The value of *hashedpassword* and *homedir* are used to verify the given username. In a MIME formatted message, the first blank line (CRLF) separates the RFC822 headers and the body of the mail message. Thus, this function returns all the characters before the first blank line in the message file.

iemsc_getmessagesource

Parameters:

String username,
String hashedpassword,
String homedir,
String foldername,
Long uid

Returns: On success, a string that contains the complete message content of the given uid, otherwise a normal IEMSC_ERR code.

Description:
This function is used to read the entire message content (including headers and all of the mail body parts) for a given uid. The value of *hashedpassword* and *homedir* are used to verify the given username.

iemsc_getembeddedheaders

Parameters:

String username,
String hashedpassword,
String homedir,
String foldername,
Long uid,
Long partnumber

Returns: On success, an associative array with the following keys:

```
hash['from']    /* String, the embedded From header */
hash['to']      /* String, the embedded To header */
hash['cc']      /* String, the embedded Cc header */
hash['date']    /* String, the embedded Date header */
```

hash['subject'] /* String, the embedded Subject header */
On failure, a normal IEMSC_ERR code is returned.

Description:
This function is used to read the 5 header fields of an embedded RFC822 MIME entity. In the example in Figure 12, we have two MIME entities. The first one is MESSAGE/RFC822 with *partnumber* equal to 0 and the second one is TEXT/HTML with *partnumber* equal to 1. We can use the *iemsc_getembeddedheaders* function to read the embedded message header (ie. From: peter@company.com ..) by setting *partnumber* to 0.

Note: *The embedded headers always belong to the MESSAGE/RFC822 MIME entity with partnumber equal 0 in this sample message.*

Message Submission

iemsc_composemail

Parameters:
String username,
String hashedpassword,
String homedir,
String charset,
String to,
String cc,
String bcc,
String subject,
String mailbody,
bool isHtmlBody,
Array attachment,
bool bSaveToDraft,
bool bSaveToOutbox

Returns: IEMSC_NO_ERR on success

Description:
This function is used to submit a message into IEMS MQ subsystem (Message Transfer Agent). The value of *hashedpassword* and *homedir* are used to verify the given username. The usage of each parameter is listed below:

charset: Defines the charset parameter for writing the mail body text. If this is set to EMPTY STRING, the charset value specified in the *iemsc_authenticate* function is used.

to, **cc**, **bcc:** Defines the recipient address in the *to*, *cc* and *bcc* list. If any of these lists contain more than one email address, a comma is used to separate each of them. Use an EMPTY STRING if you do not have any email address for that given list.

Note: *At least one email address must be present in either the to, cc or bcc lists. If no recipient address are present for any of these lists, IEMSC_ERR_NO_RECIPIENT will be returned.*

subject: Defines the message subject line. If the subject line contains non-ASCII characters, the function will encode the subject line based on the encoding scheme specified in RFC2047.

mailbody: Defines the mail content. If mailbody is an HTML formatted byte stream, set *isHtmlBody* to true. In this case, the function generates a TEXT/HTML body instead of TEXT/PLAIN body.

isHtmlBody: When set to true, the function generates a TEXT/HTML instead of a TEXT/PLAIN MIME body in the message.

attachment: An array of attachments to be added in the mail message. Each element is an associative array contain the following keys:

```
attach['ct']          /* content type */
attach['cst']         /* content subtype */
attach['cte']         /* content transfer encoding */
attach['filename']    /* the physical filename where the raw data of
                         this attachment is stored */
attach['displayname'] /* the name to be displayed by the end user
                         mail agent (ie. the name parameter in the
                         content-type header */
```

Note: *On Linux based systems, please make sure that the filename is in a directory where the IEMSC API library has read permission.*

bSaveToDraft: When set to true, the constructed message is saved to the *drafts* folder instead of being submitted to the MQ subsystem.

bSaveToOutbox: When set to true, a copy of the sent message is copied to the *outbox* folder. If *bSaveToDraft* folder is true, this flag is ignored.

Example1: Submit a simple message with no attachment:

```
$attach=array(); /* prepare an empty attachment array */
$ret = iemsc_composemail("user@company.com",
           "A456718183803",
           "c:\iems\msgstore\user@company.com",
           "us-ascii",
           "mary@company.com",
           "",
           "",
           "Hi mary!\r\nhow are you doing?\r\n",
           false,
           $attach,
           false, /* don't save it to draft folder */
           true); /* but save it to outbox */
```

Example 2: Submit a message with multiple recipients and 2 attachments:

```
$att1['ct']="application";
$att1['cst']="msword";
$att1['cte']="base64";
$att1['filename']="c:\temp\a.dat";
$att1['displayname']="myreport.doc";
$att2['ct']="image";
$att2['cst']="jpeg";
```

```
$att2['cte']="base64";
$att2['filename']="c:\temp\b.dat";
$att2['displayname']="map.jpg";
$attach=array($att1, $att2);
$ret = iemsc_composemail("user@company.com",
           "A456718183803",
           "c:\iems\msgstore\user@company.com",
           "us-ascii",
           "mary@company.com, jon@company.com",
           "",
           "boss@company.com", /* bcc to my boss ! */
          "Hi mary/jon!\r\nHere is the report of this week.\r\n",
           false,
           $attach,
           false,  /* don't save it to draft folder */
           true);  /* but save it to outbox */
```

Other Functions

iemsc_isread

Parameter:
Long flag

Returns: true if the *SEEN* bit in flag is set, false otherwise

Description:
This function checks if the *SEEN* bit in flag is set.

iemsc_isspecialfolder

Parameters:
String foldername

Returns: true if the given folder name is a *SPECIAL* folder

Description:
There are 4 special folders in the IEMS Message Store that should be not deleted or renamed. Applications can use this function to test if the folder name is one of these 4 special folders. The 4 special folder names are *inbox*, *outbox*, *trash* and *drafts*.

iemsc_utf7_to_decimal

Parameters:
String foldername

Returns: String containing a &#DDDDD; encoded string for the given folder name

Description:
This function converts the modified UTF7 encoded folder name into &#DDDDD; Many browsers are not capable to interpret the modified UTF7encoded string, however the &#DDDDD; presentation is supported by

most. Applications are suggested to use this function to convert any folder name. If the given folder name contains pure ASCII characters, no conversion will be taken.

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

Client API Constants

UID Sort Fields

Value	Description
IEMSC_SORT_BY_DEFAULT_KEY	Use the default sort field specified in the *iemsc_authenticate* function. Defaults to IEMSC_SORT_BY_UID
IEMSC_SORT_BY_UID	Sort by UID Field
IEMSC_SORT_BY_FROM	Sort by Message From Header Field
IEMSC_SORT_BY_TO	Sort by first address in the To Header Field
IEMSC_SORT_BY_SUBJECT	Sort by message Subject Header Field
IEMSC_SORT_BY_DATE	Sort by message Date Field
IEMSC_SORT_BY_SIZE	Sort by Message Size

UID Sort Order

Value	Description
IEMSC_DEFAULT_SORT_DIRECTION	Use the default sort direction specified in the *iemsc_authenticate* function. This defaults to IEMSC_SORT_DIRECTION_DECENDING.
IEMSC_SORT_DIRECTION_ASCENDING	Sort UID in ascending order
IEMSC_SORT_DIRECTION_DECENDING	Sort UID in decending order

UID Search Field

Value	Description
IEMSC_SEARCH_BY_FROM	Search UID with matched keyword in From header
IEMSC_SEARCH_BY_TO	Search UID with matched keyword in To header
IEMSC_SEARCH_BY_SUBJECT	Search UID with matched keyword in Subject header
IEMSC_SEARCH_BY_DATE	Search UID which matches a date or date range

Client API Error Codes

Error Code	Description
IEMSC_NO_ERR	No error
IEMSC_MQ_INIT_FAIL	Unable to initialize the MQ subsystem
IEMSC_INVALID_PARAM	Invalid Paramater Passed to Function
IEMSC_INVALID_PART_NUMBER	Part Number Invalid for Given UID
IEMSC_ERR_NO_RECIPIENT	No Recipient Addresses Present
IEMSC_ERR_DECODE_STRING	Unable to Decode MIME Body Part
IEMSC_SPECIAL_FOLDER	Given Folder is a Special Folder
IEMSC_ERR_OPEN_MESSAGE_FILE	Cannot Open Message File for Read Access. File may be missing or IEMSCPHP extension does not have READ permission on the message file.
IEMSC_ERR_WRITE_MESSAGE_FILE	Unable to write data to the message file. The IEMSCPHP extension may not have WRITE permission on the message file or disk full.
IEMSC_ERR_READ_MESSAGE_FILE	Unable to read the message file. The file may be corrupted or missing.
IEMSC_ERR_SUBMIT_MESSAGE	Unable to submit the constructed message to the MQ subsystem.
IEMSC_FAIL_CREATE_TMP_FILE	Unable to create temporary file. The IEMSCPHP extension may not have WRITE permission in the IEMS temporary directory.
IEMSC_ERR_WRITE_TEMP_FILE	Unable to write data to the temporary file. The IEMSCPHP extension may not have WRITE permission to the temporary file or disk full.
IEMSC_UID_NOT_FOUND	The given UID cannot be found in the folder
IEMSC_FOLDER_ALREADY_EXISTS	The given folder name already exists
IEMSC_ERR_MESSAGE_NOT_FOUND	The message file cannot be found
IEMSC_FOLDER_NOT_EMPTY	The given folder is not empty - removal failed.
IEMSC_FOLDER_NOT_EXISTS	The given folder name does not exist.
IEMSC_FAIL_CREATE_FOLDER	Unable to create the new folder. The IEMSCPHP extension may not have WRITE permission in the IEMS Message Store subsystem.
IEMSC_FAIL_RENAME_FOLDER	Unable to rename the folder. The IEMSCPHP extension may not have WRITE permission in the IEMS Message Store subsystem.
IEMSC_INVALID_FOLDER_NAME	The given folder name is not in a valid format.
IEMSC_NOT_AUTHENTICATED	The given username is not authenticated.
IEMSC_ERR_INVALID_USER	The given username is not valid
IEMSC_ERR_INVALID_PWD	The given clear text password is not valid
IEMSC_ERR_INTERNAL_ERR	Internal Error - the system may have run out of memory or other error in Message Store subsystem.
IEMSC_ERR_READ_ATTACHMENT	Unable to read the file attachment when composing new mail message. The IEMSCPHP extension may not have READ permission for the file specified in the *filename* passed value.

Error Code	Description
IEMSC_SESSION_EXPIRED	The login session has expired. The application should direct the user to re-login.
IEMSC_INVALID_SESSION	The hashed password is not valid
IEMSC_ERR_INVALID_PASSWORD_FORMAT	The new password is not valid. The new password must contain at least 6 characters.
IEMSC_ERR_UPDATE_PASSWORD	System is unable to update the new password with the IEMS Directory.

APPENDIX D

Message Store Naming Issues

The IEMS Message Store subsystem uses a modified UTF7 encoding scheme for storing file names. This naming convention is documented in section 5.1.3 of RFC2060. It is used to store folder name in international languages such as Chinese and other non-ASCII character sets. For example, the Chinese words

means 'China'. The UNICODE values are *u20013 u22283* respectively. When converted to UTF7 encoding, this folder name becomes *&Ti1XCw-*.

The IEMSCPHP extension functions always express folder name parameter encoded in this UTF7 format, with the exception of the *iemsc_renamefolder* and *iemsc_createfolder* functions. In these two functions, the newfoldername parameter should be encoded in UTF8 format. In this example, the UTF8 byte stream for 'China' is *0xE4B8AD 0xE59C8B* respectively. These two functions will convert the UTF8 encoded folder name into UTF7 format on behalf of the application.

Note: *Web programmer can make use of the 'charset=utf-8' parameter specified in the META header. The input folder name is converted into UTF8 format by the browser before the value is submitted to the Web server. To achieve this result, put the following META header in your HTML page:*

```
<META HTTP-EQUIV=CONTENT-TYPE CONTENT="TEXT/HTML CHARSET=UTF-8">
```

As the modified UTF7 encoding scheme is not supported by many currently available browsers, it is best to encode file names using a decimal encodede version of the Unicode character. The iemsc_utf7_to_decimal routing can be used to convert UTF7 encoded folder name into this *&#DDDDD;* representation. The *&#DDDDD;* representation is supported by most of the modern browsers including Internet Explorer 5, and 6 and Netscape 6.x and 7.x.

APPENDIX E

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

I

L

M

N

P

Q

R

S

U

V

W

Z