Oracle® eMail Server

Developer's Guide

Release 5.2

Part No. A86650-01

January, 2001

ORACLE

Oracle eMail Server Developer's Guide , Release 5.2

Part No. A86650-01

Primary Author: Ginger Tabora

Contributors: Vikas Dhamija, Sandra Lee, Anthony Ye

Contents

Send Us Your Comments vii

Preface ix

1 Oracle eMail Server API Overview

PL/SQL APIs 1-2
IOSend API 1-2
IM_API 1-2
Server Side Rules 1-2
DAPLS API 1-3

2 Oracle eMail Server PL/SQL API Reference

Using the IOSend Package 2-2
Prerequisites and Setup 2-2
IOSend Process Overview 2-4
IOSend Procedure Reference 2-5
Example 1: Sending a Simple Message 2-14
Example 2: Sending a MIME Message with a Binary Attachment 2-16
Using the IM_API Package 2-20
Prerequisites and Setup 2-20
IM_API Function Reference 2-21
Authentication and Initialization Functions 2-22
Logging Functions 2-23
Folder Management Functions 2-25
Message Generation Functions 2-29
Checking New Mail Functions 2-34
Message Navigation Functions 2-36
Message Foldering Functions 2-40
Message Property Functions 2-45
Message Parsing Functions 2-50

IM_API Usage Sample Script 2-59
Using Server Side Rules 2-68
Understanding Mail Rules 2-68
Rule Management API 2-68
User Rules Management API Function Reference Object Types 2-69
Functions 2-72
Administrator Rules Management API Function Reference 2-77
Viewing a List of Rules 2-84
Writing Customized Procedure for ACTION_RUN_PROC 2-85
Server Side Rules Sample Script 2-85
Using the DAPLS Package 2-93
Prerequisites and Setup 2-93
DAPLS Procedure Reference 2-94

Index

Send Us Your Comments

Oracle Oracle eMail Server Developer's Guide, Release 5.2

Part No. A86650-01

Oracle Corporation welcomes your comments and suggestions on the quality and usefulness of this publication. Your input is an important part of the information used for revision.

- Did you find any errors?
- Is the information clearly presented?
- Do you need more information? If so, where?
- Are the examples correct? Do you need more examples?
- What features did you like most about this manual?

If you find any errors or have any other suggestions for improvement, please indicate the chapter, section, and page number (if available). You can send comments to us in the following ways:

- Electronic mail: infodev@us.oracle.com
- FAX: (650) 506-7228 Attn: Oracle Oracle eMail Server Documentation Manager
- Postal service:
  Oracle Corporation
  Oracle eMail Server Documentation Manager
  500 Oracle Parkway, Mailstop 4OP12
  Redwood Shores, CA 94065
  USA

If you would like a reply, please give your name, address, and telephone number below.

If you have problems with the software, please contact your local Oracle Support Services representative.

Preface

This preface includes the following topics:

- Intended Audience
- Typographic Conventions
- Oracle Oracle eMail Server Documentation

Intended Audience

This *Developer's Guide* primarily addresses the application developer audience. It provides an introduction to Oracle eMail Server and describes the management tasks you will perform as an Oracle eMail Server server administrator. PL/SQL programming knowledge is also helpful in implementing PL/SQL APIs and server-side rules.

Typographic Conventions

The following typographic conventions are used in this manual:

Convention	Description
italic	Italicized type identifies document titles.
`Monospace`	Monospace type indicates commands.
bold	Boldface type indicates script names, directory names, path names, and file names (for example, the **root.sh** script).
UPPERCASE	Uppercase letters indicate parameters or environment variables (for example, ORACLE_HOME).
. . .	In code examples, vertical ellipsis points indicate that information not directly related to the example has been omitted.
. . .	In command syntax, horizontal ellipsis points indicate repetition of the preceding parameters. The following command example indicates that more than one input_file may be specified on the command line: `command [input_file ...]`
< >	In command syntax, angle brackets identify variables that the user must supply. You do not type the angle brackets. The following command example indicates that the user must enter a value for the variable input_file: `command <input_file>`
[ ]	In command syntax, brackets enclose optional clauses from which you can choose one or none. You do not type the brackets. The following command example indicates that the variable output_file is optional: `command <input_file> [output_file]`

Convention	Description
{ }	In command syntax, curly brackets indicate that a choice of two or more items separated by a vertical bar or pipe (|). You do not type the curly brackets. The following command example indicates a choice of either a or b: `command {a | b}`
$	The dollar sign represents the shell prompt in UNIX. [1]

[1] In examples, an implied carriage return occurs at the end of each line, unless otherwise noted. You must press the [Enter] key at the end of a line of input.

Oracle Oracle eMail Server Documentation

Oracle Oracle eMail Server documentation is available in HTML and PDF format on the CD-ROM and installs automatically during product installation. Use your Web browser to access **$ORACLE_HOME/doc/es52/index.htm** on your server. The following documents are available:

- *Oracle eMail Server Release Notes*
- *Oracle eMail Server Administrator's Guide*
- *Oracle eMail Server Installation Guide*

1

Oracle eMail Server API Overview

The Oracle eMail Server PL/SQL APIs (application program interfaces) can be used to access and extend the core functionality of Oracle eMail Server. The APIs included in this manual provide a means of exposing Oracle eMail Server functionality that can then be customized to suit your business and application requirements.

PL/SQL APIs

Oracle eMail Server's PL/SQL APIs (application program interfaces) provide the interfaces to customize the Oracle eMail Server's functionality. PL/SQL packages for this release include the following:

- IOSend API
- IM_API
- Server Side Rules

IOSend API

The `IOSend` package provides the functions necessary to send MIME (Multipurpose Internet Mail Extension) messages with Oracle eMail Server. This package can be called to compose messages that conform to the MIME standard, including simple text messages or messages with attachments.

> **See Also:** "Using the IOSend Package" on page 2-2 for detailed reference information and sample scripts

IM_API

The `IM_API` package provides access to the Oracle eMail Server message store, allowing PL/SQL programmers to retrieve, manage, and send e-mails with an Oracle eMail Server mail account.

> **See Also:** "Using the IM_API Package" on page 2-20 for detailed reference information and sample scripts

Server Side Rules

The `Server Side Rules` package provides the functions necessary to define automatic actions to perform on incoming messages when the messages meet custom, predefined criteria. For example, you can define a rule to delete incoming messages that contain a specified string in the Subject field.

> **See Also:** "Using Server Side Rules" on page 2-68 for detailed reference information and sample scripts

DAPLS API

The DAPLS package provides procedures which enables some, but not all OOMGR commands to be performed from your PL/SQL code. The package can be called to create and delete user accounts, create and maintain public distribution lists, and perform a wide variety of tasks.

> **See Also:** "Using the DAPLS Package" on page 2-93 for detailed reference information and sample scripts

2

Oracle eMail Server PL/SQL API Reference

This chapter describes the PL/SQL APIs for Oracle eMail Server. Topics include the following:

- Using the IOSend Package
- IOSend Procedure Reference
- Using the IM_API Package
- IM_API Function Reference
- Using Server Side Rules
- Administrator Rules Management API Function Reference
- Using the DAPLS Package

Using the IOSend Package

A PL/SQL package named `IOSend` is installed as part of Oracle eMail Server to enable PL/SQL programmers to send Multipurpose Internet Mail Extension (MIME) messages using Oracle eMail Server. This package can be called to compose any messages that conform to the MIME standard, including simple text messages or messages with attachments.

An e-mail message is said to be MIME compliant if the message contains headers defined in the MIME standard. The MIME standard defines a mechanism to represent a complex message in plain text using headers, boundaries, and encoding methods. MIME enables messages that contain non-textual data, one or more message attachments, or messages written in a non-English character set to be transmitted using Simple Mail Transfer Protocol (SMTP), the standard for Internet e-mail transport.

In order to accommodate sending messages with complex MIME structures, this package exposes a calling interface that requires the caller to manually assemble a message part by part. Therefore, an understanding of MIME message structure is required for using this tool.

Prerequisites and Setup

The `IOSend` package is loaded with installation of Oracle eMail Server. In addition, any application that uses to use this tool must perform the following initial setup steps:

1. The API requires the user to be authenticated by the Oracle eMail Server before sending any e-mail. Although the user can use the API as any database user, the user still must have an Oracle eMail Server user account. If such an account is not present, then an administrator can create it for the user by using the Oracle eMail Server Java Administration Tool.
2. (Optional) For convenience, a synonym can be created for the database user who uses this API so that a program can directly refer to the `IOSend` package without owner qualification. This can be done in SQL*Plus for any database user as follows:

   ```
   SQL>create synonym iosend for oo.iosend;
   ```

   See Also: *Oracle eMail Server Administrator's Guide* for more information on how to create user accounts

3. This API uses Oracle PL/SQL External Procedure feature internally, make sure external procedure agent is set up properly. If the server was migrated from a version earlier than Oracle 8 or if the Net8 component is custom configured, the administrator may need to manually setup the environment to allow external procedure call.

   See Also: *Oracle 8i Administrator's Guide* for more information on managing Oracle processes

IOSend Process Overview

In order to send a MIME message, the following steps should be performed:

1. Authenticate the caller as a valid Oracle eMail Server user.
2. Start a message by providing envelope information. In this step the user specifies message recipient, originator, subject, priority, and expiration date.
3. (Optional) Add extra custom-defined headers. For example, you can add a Phone header to the message.
4. (Optional) Provide MIME structural header for the MIME message. This is only required if the message contains more than one part. For example, a message with a file attachment is a multipart message. The most frequently used MIME structural header is multipart/mixed for messages with attachments.
5. (Optional) Describe the message content you are about to add to the message. If the message part is a file attachment, then describing the part includes providing attachment type and file name, along with other attributes.
6. (Optional) Add additional MIME attributes if necessary. This can include headers such as content disposition or content transfer encoding.
7. Add the message content that you described in step 5.
8. (Optional) Repeat steps 5-7 if the message contains more than one part. For example, an e-mail message with a file attachment requires two rounds of MIME part processing.
9. (Optional) Specify a folder carbon copy destination. For example, you can make a copy of your message in a folder called Sent Messages.
10. Complete and commit the message submission to make it available to the transport agent. This is when the message starts to get delivered.

Note: Each of the above steps corresponds to one procedure in the package.

IOSend Procedure Reference

The following procedures are described in this section:

- AUTHENTICATE Procedure
- SUBMIT_HEADER Procedure
- ADD_EXTRA_HEADER Procedure
- ADD_MIME_MARKER Procedure
- DESCRIBE_PART Procedure
- ADD_ATTRIBUTE Procedure
- ADD_CONTENT Procedure
- FOLDER_CARBON_COPY Procedure
- SUBMIT_MESSAGE Procedure

AUTHENTICATE Procedure

The `AUTHENTICATE` procedure should always be called first to start an `IOSend` session. You cannot send a message using `IOSend` without being authenticated as a valid Oracle eMail Server user.

Syntax for the AUTHENTICATE Procedure

```
PROCEDURE AUTHENTICATE(
    username  IN   VARCHAR2,
    domain    IN   VARCHAR2,
    password  IN   VARCHAR2,
    status    OUT  INTEGER);
```

Input for the AUTHENTICATE Procedure

Input	Description
`username`	The Oracle eMail Server account user name
`domain`	The fully qualified domain name for the user. It can be `NULL` if the user is unique on a node.
`password`	The unencrypted password to authenticate the user

Output of the AUTHENTICATE Procedure

The `status` is returned with one of the following values (all calls use the same set of output statuses):

Table 2–1

Status	Description
0	Successful
1	No such user
2	Ambiguous user
3	Action out of order
4	Invalid priority
5	Invalid message type
6	Missing recipients
7	Invalid attachment marker
8	Invalid binary flag
9	Invalid folder in FCC:
10	Invalid attachment type

SUBMIT_HEADER Procedure

The SUBMIT_HEADER procedure is called to submit message envelope information.

Syntax for the SUBMIT_HEADER Procedure

```
PROCEDURE SUBMIT_HEADER(
tostr     IN   VARCHAR2,
ccstr     IN   VARCHAR2,
bccstr    IN   VARCHAR2,
subj      IN   VARCHAR2,
replyto   IN   VARCHAR2,
fromstr   IN   VARCHAR2,
defdate   IN   VARCHAR2,
expdate   IN   VARCHAR2,
priority  IN   INTEGER,
sflags    IN   INTEGER,
status    OUT  INTEGER);
```

Input for the SUBMIT_HEADER Procedure

Table 2–2

Input	Description
`tostr`, `ccstr` and `bccstr`	These parameters cannot all be `NULL` arguments. At least one recipient must be specified. You can specify multiple recipients by giving a composite argument to `tostr`, `ccstr`, or `bccstr`. For example, a possible `tostr` argument can be `'bwayne@gotham.com, dgrayson@gotham.com'`.
`subj`	The subject of the message
`replyto`	The Reply-to header of the message. This header is optional. If `replyto` is not specified, then all replies to this e-mail are directed to the caller's Oracle eMail Server user mailbox.
`fromstr`	Typically the full name of the e-mail sender, but it can be any name designated by the caller.
`defdate`	The deferred delivery time, you can delay message delivery by setting this time to a later time than the current time. All date strings must be in Oracle date format: DD-MM-YYYY HH24:MM:SS.
`expdate`	The expiration date of this message. If `expdate` is not set, the message does not expire by itself. A message marked for expiration will be deleted from the system automatically after the expiration date. The feature is only useful when the message is sent to an Oracle eMail Server user.
`priority`	This parameter should be one the three values –50, 0, 50, representing low, normal, and high priority, respectively.

`sflags`	This parameter can be any or a combination of the following four values:
	`IOSEND_HFLG_NOAREPLY`
	`IOSEND_HFLG_NOAFWD`
	`IOSEND_HFLG_NONDR`
	`IOSEND_IOSEND_HFLG_REALATT`
	For example, a message with a file attachment that does not require any delivery report on bad recipients should set `sflags` to `IOSEND_IOSEND_HFLG_REALATT + IOSEND_HFLG_NONDR`.

Output of the SUBMIT_HEADER Procedure

The `status` returned is one of the values listed under "Output of the AUTHENTICATE Procedure" on page 2-6.

ADD_EXTRA_HEADER Procedure

This function enables you to add customized headers to the message. Conventionally, non-standard headers, as opposed to headers registered in the MIME standard, should have a prefix of "X-". If you want to add a header that includes the sender's phone number, the header should look like "X-Phone:" instead of "Phone:". Headers submitted that do not following this convention will be prefixed by "X-ORCL-APPLICATION:" automatically by the server.

Syntax for the ADD_EXTRA_HEADER Procedure

```
PROCEDURE ADD_EXTRA_HEADER(
    lhs     IN   VARCHAR2,
    rhs     IN   VARCHAR2,
    ono     IN   INTEGER,
    status  OUT  INTEGER);
```

Input for the ADD_EXTRA_HEADER Procedure

Table 2–3

Input	Description
`lhs (left-hand side)`	This parameter is the name of the header. It should include a colon at the end. For example, if you want to add a phone header to the message, then `lhs` should be "X-Phone:".
`rhs (right-hand side)`	This parameter is the value of the header. For example, "1-800-555-1212" would be a value for a phone header.
`ono`	Order number has to be kept by the caller. If you have multiple headers to add, then order number should start from 1 and increment by 1 every time a new header is added.

Output of the ADD_EXTRA_HEADER Procedure

The `status` returned is one of the values listed under "Output of the AUTHENTICATE Procedure" on page 2-6.

ADD_MIME_MARKER Procedure

The most common use for the ADD_MIME_MARKER call is to establish a multipart MIME structure. For example, if you want to send messages with attachments, then you must call this procedure with marker_name being set to 'multipart/mixed'.

Syntax for the ADD_MIME_MARKER Procedure

```
PROCEDURE ADD_MIME_MARKER(
    marker_name  IN   VARCHAR2,
    mlevel       IN   VARCHAR2,
    status       OUT  INTEGER);
```

Input for the ADD_MIME_MARKER Procedure

Table 2–4

Input	Description
marker_name	MIME structural header
mlevel	A numbered tag indicating the relative position of this multipart construct with respect to the whole MIME message. This allows for hierarchically organized MIME messages with multiple levels of 'multipart/mixed' construct. In most cases, this should be 0 to indicate that it is a top level MIME construct.

Output of the ADD_MIME_MARKER Procedure

The status returned is one of the values listed under "Output of the AUTHENTICATE Procedure" on page 2-6.

DESCRIBE_PART Procedure

DESCRIBE_PART is used to describe an attachment. This is where you provide information such as content type, file name, and size of the attachment before you submit an attachment. This information is essential since MIME requires every attachment to have headers containing properties of the attachment.

Syntax for the DESCRIBE_PART Procedure

```
PROCEDURE DESCRIBE_PART(
    filename   IN   VARCHAR2,
    descr      IN   VARCHAR2,
    mime_type  IN   VARCHAR2,
```

```
    pflags     IN   INTEGER,
    mlevel     IN   VARCHAR2,
    status     OUT  INTEGER);
```

Input for the DESCRIBE_PART Procedure

Table 2–5

Input	Description
`filename`	The attachment filename
`descr`	A description of the attachment
`mime_type`	This parameter can be any of the valid MIME content types. The following are commonly used MIME content types: ■ text / plain (plain text) ■ application / octet-stream (untyped binary) ■ text / html (HTML documents) ■ audio / basic (audio data) ■ image / tiff (Tagged Image File Format) ■ image / gif (Graphics Interchange Format) ■ image / g3fax (G3 FAX image) ■ image / x-pict (PICT image) ■ image / jpeg (JPEG image) ■ image / x-xbitmap (X bitmap) ■ application / postscript (PostScript) ■ application / zip (ZIP archive) ■ video / quicktime (Quicktime video) ■ video / mpeg (MPEG video) ■ application / wordperfect5.1 (WordPerfect) ■ application / msword (Microsoft Word) ■ application / pdf (Adobe Acrobat)
`pflags`	The same range as `aflags` in the "ADD_CONTENT Procedure" on page 2-12, except the `IOSEND_AFLG_DONE` is not used. Therefore `IOSEND_AFLG_PRIMARY` and `IOSEND_AFLG_HIDDEN` are valid.

Table 2–5

Input	Description
mlevel	The same meaning as mlevel in "ADD_MIME_MARKER Procedure" on page 2-9. Typically, for a simple message body, it should be set to 0. For a message with attachments, the main message body should have level 1, the first attachment should have level 2, and so on.

Output of the DESCRIBE_PART Procedure

The status returned is one of the values listed under "Output of the AUTHENTICATE Procedure" on page 2-6.

ADD_ATTRIBUTE Procedure

This procedure is used to add additional attributes to the message part you added by calling ADD_CONTENT. Typically additional attributes include only MIME attributes like content-transfer-encoding or content-disposition.

Syntax for the ADD_ATTRIBUTE Procedure

```
PROCEDURE ADD_ATTRIBUTE(
    attr_name   IN   VARCHAR2,
    attr_value  IN   VARCHAR2,
    amarker     IN   INTEGER,
    aflags      IN   INTEGER,
    mlevel      IN   VARCHAR2,
    status      OUT  INTEGER);
```

Input for the ADD_ATTRIBUTE Procedure

Table 2–6

Input	Description
attr_name	The MIME attribute name
attr_value	The MIME attribute value. For example, one possible value for "Content-Disposition" can be "attachment."

Table 2–6

Input	Description
amarker	This parameter should be set to one of the following values: IOSEND_ATT_STRUCT IOSEND_ATT_PRIMARY IOSEND_ATT_INCLID IOSEND_ATT_REALATT Only values IOSEND_ATT_PRIMARY or IOSEND_ATT_REALATT are relevant in this procedure depending on whether you are adding attributes for the main message body or an attachment.
aflags	This parameter takes the same values as aflags in the "ADD_CONTENT Procedure" on page 2-12, except that IOSEND_ATT_PRIMARY is irrelevant here.
mlevel	This parameter should be the same value passed in "ADD_MIME_MARKER Procedure" on page 2-9.

Output of the ADD_ATTRIBUTE Procedure

The status returned is one of the values listed under "Output of the AUTHENTICATE Procedure" on page 2-6.

ADD_CONTENT Procedure

This call can be made several times to complete one large message part.

Syntax for the ADD_CONTENT Procedure

```
PROCEDURE ADD_CONTENT(
    vbuff   IN   VARCHAR2,
    rbuff   IN   RAW,
    buflen  IN   INTEGER,
    aflags  IN   INTEGER,
    status  OUT  INTEGER);
```

Input for the ADD_CONTENT Procedure

Table 2–7

Input	Description
vbuff	Contains one piece of a textual message part. It should either be a mail body or a textual attachment body.
rbuff	Contains one piece of a binary message part. It should be a binary mail body or a binary attachment body.
buflen	This parameter should be the length of the text buffer
aflags	An input parameter that should be any or a combination of the following three values: IOSEND_AFLG_PRIMARY IOSEND_AFLG_DONE IOSEND_AFLG_MIMEATTR For example, if aflags has a value of IOSEND_AFLG_PRIMARY + IOSEND_AFLG_DONE, then that indicates that the whole message body is in vbuff and this is not an attachment. This should be the most commonly used flag value.

Output of the ADD_CONTENT Procedure

The status returned is one of the values listed under "Output of the AUTHENTICATE Procedure" on page 2-6.

FOLDER_CARBON_COPY Procedure

This procedure saves the current outgoing message to a folder. The folder must be owned by the user currently authenticated for the message composition.

Syntax for the FOLDER_CARBON_COPY Procedure

```
PROCEDURE FOLDER_CARBON_COPY(
    fname   IN   VARCHAR2,
    fid     IN   INTEGER,
    status  OUT  INTEGER);
```

Input for the FOLDER_CARBON_COPY Procedure

Table 2–8

Input	Description
`fname`	Name of the folder to which you want to copy the message. The folder name is case sensitive and must be in the form of an absolute folder path name. If you want to specify a folder named "Sent Items", then you should pass the value "/Sent Items" with the leading "/" character.
`fid`	Folder ID of the folder into which you want to save the message. This folder must be a private folder owned by the calling user's corresponding Oracle eMail Server account. ■ If `fid` is `0`, then `fname` is used to determine the FCC folder. ■ If `fid` is not `0`, then `fname` is ignored.

Output of the FOLDER_CARBON_COPY Procedure

The `status` returned is one of the values listed under "Output of the AUTHENTICATE Procedure" on page 2-6.

SUBMIT_MESSAGE Procedure

This is the final procedure to call before issuing a commit. This call makes the message available for transport.

Syntax for the SUBMIT_MESSAGE Procedure

```
PROCEDURE SUBMIT_MESSAGE(
    status  OUT  INTEGER);
```

Output of the SUBMIT_MESSAGE Procedure

The `status` returned is one of the values listed under "Output of the AUTHENTICATE Procedure" on page 2-6.

Example 1: Sending a Simple Message

This example demonstrates a simple PL/SQL script used to send a simple plain text message to a recipient. To run this script, modify the constant declaration section to make sure they contain valid values. The example can be accessed in the following directory:

$ORACLE_HOME/office/admin/samples/iosend_sample1.sql

Example

```
DECLARE
    v_status INTEGER := 0; -- return status for the iosend procedures

    -- constant values used in this example
    c_user CONSTANT VARCHAR2(80) := 'scott';
    c_password CONSTANT VARCHAR2(80) := 'tiger';
    c_domain CONSTANT VARCHAR2(80) := 'acme.com';
    c_tostr CONSTANT VARCHAR2(80) := 'jdoe@acme.com';
    c_subject CONSTANT VARCHAR2(80) := 'Hello World';
    c_body CONSTANT VARCHAR2(512) :=
        'John,' || chr(10) || chr(10) ||
        'Just want to say hello!' || chr(10) ||
        '-- Scott';
BEGIN
    -- authentication is required to send a mail
    iosend.authenticate(c_user, c_domain, c_password, v_status);
    IF v_status != 0 THEN
        dbms_output.put_line('AUTHENTICATE user error ' || v_status);
        RETURN;
    END IF;

    -- start the message by submitting the headers first
    -- non-applicable headers will just be NULL
    iosend.submit_header(c_tostr, NULL, NULL, c_subject, NULL, NULL,
        NULL, NULL, 0, 0, v_status);
    IF v_status != 0 THEN
        dbms_output.put_line('Submit header error ' || v_status);
        rollback;
        RETURN;
    END IF;

    -- now submit the message body
        iosend.add_content(c_body, NULL, length(c_body),
        iosend.iosend_aflg_primary + iosend.iosend_aflg_done, v_status);
    IF v_status != 0 THEN
        dbms_output.put_line('Submit content error ' || v_status);
        rollback;
        RETURN;
    END IF;

    -- finally submit the message
```

```
    iosend.submit_message(v_status);
IF v_status != 0 THEN
    rollback;
    RETURN;
END IF;

-- time to commit;
    commit;
END;
```

Example 2: Sending a MIME Message with a Binary Attachment

This script sends a message with its main body in HTML The content is loaded from an HTML file using DBMS_LOB package supplied by Oracle RDBMS. The message also has a GIF attachment that is loaded from another file using DBMS_ LOB package.

Before running this example script, you need to perform the following steps:

1. Make sure the database account that you run this script from has the privilege to CREATE ANY DIRECTORY and DROP ANY DIRECTORY. If not, have them granted by DBA.
2. Copy an HTML document and a JPG image file to the directory /tmp and make sure they are readable
3. Edit the constant section at the top of this script to make sure usernames, password, domain and filenames are all valid.

This example can be accessed in the following directory:

`$ORACLE_HOME/office/admin/samples/iosend_sample2.sql`

Example

```
DROP directory picdir;
CREATE directory picdir AS '/tmp';

DECLARE
    v_status INTEGER; -- return status from iosend functions
    v_rawbuf RAW(2400);
    v_done BOOLEAN := false;
    v_flag INTEGER;
    v_bfile bfile;
    v_amt INTEGER;
    v_offset INTEGER := 1;
```

```
    c_buflen CONSTANT INTEGER := 2400;
    c_user CONSTANT VARCHAR2(80) := 'scott';
    c_domain CONSTANT VARCHAR2(80) := 'acme.com';
    c_password CONSTANT VARCHAR2(80) := 'tiger';
    c_tostr CONSTANT VARCHAR2(80) := 'tom@acme.com';
    c_cc CONSTANT VARCHAR2(80) := 'dick@acme.com';
    c_subject CONSTANT VARCHAR2(80) := 'IOSEND Test Mail';
    c_fromstr CONSTANT VARCHAR2(80) := 'Sample User';
    c_replyto CONSTANT VARCHAR2(80) := 'harry@acme.com';
    c_dirname CONSTANT VARCHAR2(40) := 'PICDIR';
    c_bodytype CONSTANT VARCHAR2(40) := 'text/html';
    c_txtname CONSTANT VARCHAR2(80) := 'message.html';
    c_mimetype CONSTANT VARCHAR2(40) := 'image/jpg';
    c_filename CONSTANT VARCHAR2(80) := 'picture.jpg';
BEGIN
    -- authentication is required to send a mail
    iosend.authenticate(c_user, c_domain, c_password, v_status);
    IF v_status != 0 THEN
        dbms_output.put_line('AUTHENTICATE error = ' || v_status);
        RETURN;
    END IF;

    -- start the message by submitting the header
    -- this mail has both TO and CC headers, with a customized Reply-To
    -- and FROM header.  The expiration is set to be 3 weeks after it is
-- sent, the header flags indicate that no Auto-replies are
-- accepted.

    iosend.submit_header(c_tostr, c_cc, NULL, c_subject, c_replyto,
        c_fromstr, NULL, to_char(sysdate + 21, 'DD-MM-YYYY HH24:MI:SS'),
        0, iosend.iosend_hflg_noareply + iosend.iosend_hflg_realatt,
        v_status);
    IF v_status != 0 THEN
        dbms_output.put_line('Submit header error = ' || v_status);
        rollback;
        RETURN;
    END IF;

    -- now add a custom header
    --- this mail will have a custom header "X-Confidential-Level:"
    iosend.add_extra_header('X-Confidential-Level:',
        'internal-circulation', 1, v_status);
    IF v_status != 0 THEN
        dbms_output.put_line('Add header error = ' || v_status);
```

```
    rollback;
    RETURN;
END IF;

-- now copy it to a server side folder of the sender
--- copy it to the folder "Sent"
iosend.folder_carbon_copy('/Sent', 0, v_status);
IF v_status != 0 THEN
    dbms_output.put_line('FCC error = ' || v_status);
    rollback;
    RETURN;
END IF;

-- now add the MIME structural marker 'multipart/mixed'
--- add this header because the message contains an attachment
iosend.add_mime_marker('multipart/mixed', 0, v_status);
IF v_status != 0 THEN
    dbms_output.put_line('Add multipart/mixed header error = '
    || v_status);
    rollback;
    RETURN;
END IF;

-- now desribe the message body in HTML
iosend.describe_part(NULL, NULL, c_bodytype,
    iosend.iosend_aflg_primary, 1, v_status);
IF v_status != 0 THEN
    dbms_output.put_line('Describe body error = ' || v_status);
    rollback;
    RETURN;
END IF;

-- read the HTML body using DBMS_LOB from a file
v_bfile := bfilename('PICDIR', c_txtname);
dbms_lob.fileopen(v_bfile, dbms_lob.file_readonly);
LOOP
    v_amt := c_buflen;
    dbms_lob.read(v_bfile, v_amt, v_offset, v_rawbuf);
    v_offset := v_offset + v_amt;

-- setup flags to tell IOSEND whether we have multiple trips
v_flag := iosend.iosend_aflg_primary +
    iosend.iosend_aflg_mimeattr;
    IF v_amt < c_buflen THEN
v_flag := v_flag + iosend.iosend_aflg_done;
```

```
 END IF;

   -- now submit the main body
   iosend.add_content(utl_raw.cast_to_VARCHAR2(v_rawbuf), NULL, 0,
     v_flag, v_status);
   IF v_status != 0 THEN
dbms_output.put_line('Add body error = ' || v_status);
rollback;
RETURN;
   END IF;
   EXIT WHEN v_amt < c_buflen;
END LOOP;
dbms_lob.fileclose(v_bfile);

-- now describe the attachment
-- note that mlevel here is 2 and pflag is 0
iosend.describe_part(c_filename, NULL, c_mimetype, 0, 2, v_status);
IF v_status != 0 THEN
   dbms_output.put_line('Describe attachment error = ' || v_status);
   rollback;
   RETURN;
END IF;

-- now add MIME attribute content-disposition to attachment.
-- Note that amarker is 4 for MIME attribute, aflag is 4 for
-- attachment and mlevel is 2 as before
iosend.add_attribute('Content-Disposition:',
  'inline; filename="' || c_filename || '"',
  iosend.iosend_att_realatt + iosend.iosend_att_nonprint,
  iosend.iosend_aflg_mimeattr, 2, v_status);
IF v_status != 0 THEN
   dbms_output.put_line('Add attribute error = ' || v_status);
   rollback;
   RETURN;
END IF;

-- read the attachment from a file
v_bfile := bfilename(c_dirname, c_filename);
dbms_lob.fileopen(v_bfile, dbms_lob.file_readonly);
v_offset := 1;
LOOP
   v_amt := c_buflen;
   dbms_lob.read(v_bfile, v_amt, v_offset, v_rawbuf);
   v_offset := v_offset + v_amt;
```

```
      IF v_amt < c_buflen THEN
  v_flag := iosend.iosend_aflg_done;
      ELSE
  v_flag := 0;
      END IF;

      -- now submit the attachment.
      iosend.add_content(NULL, v_rawbuf, v_amt, v_flag, v_status);
      IF v_status != 0 THEN
  dbms_output.put_line('Add attachment error = ' || v_status);
  rollback;
  RETURN;
      END IF;
      EXIT WHEN v_amt < c_buflen;
   END LOOP;
   dbms_lob.fileclose(v_bfile);

   -- finally submit the message
   iosend.submit_message(v_status);
   IF v_status != 0 THEN
      dbms_output.put_line('Submit message error = ' || v_status);
      rollback;
      RETURN;
   END IF;

   -- time to commit;
   commit;
END;
```

Using the IM_API Package

The Oracle eMail Server PL/SQL mail API (`IM_API`) is an interface to the Oracle eMail Server message store that enables PL/SQL programmers to retrieve, manage, and send e-mails with an Oracle eMail Server e-mail account.

Prerequisites and Setup

The `IM_API` package is automatically loaded with the installation of Oracle eMail Server. In addition to this installation, any application that uses this package must perform the following initial setup steps:

1. The API requires the user to be authenticated by the Oracle eMail Server before sending any e-mail. Although a user can use the API as any database user, the user still must have an Oracle eMail Server user account. If such an account is

not present, an administrator can create it for the user by using the Oracle eMail Server Java Administration Tool.

2. (Optional) For convenience, a synonym can be created for the database user who uses the API so that a program can directly refer to the `IM_API` without owner qualification. This can be done in SQL*Plus for any database user as follows:

```
SQL>create synonym im_api for oo.im_api;
```

See Also: *Oracle eMail Server Administrator's Guide* for more information on how to create user accounts

3. This API uses Oracle PL/SQL External Procedure feature internally, make sure external procedure agent is set up properly. If the server was migrated from a version earlier than Oracle 8 or if the Net8 component is custom configured, the administrator may need to manually setup the environment to allow external procedure call.

See Also: *Oracle 8i Administrator's Guide* for more information on managing Oracle processes

IM_API Function Reference

IM_API functions are presented in groups with related functionality. The following groups are documented in this section:

- Authentication and Initialization Functions
- Logging Functions
- Folder Management Functions
- Message Generation Functions
- Checking New Mail Functions
- Message Navigation Functions
- Message Foldering Functions
- Message Property Functions
- Message Parsing Functions
- IM_API Usage Sample Script

Authentication and Initialization Functions

The following functions are described in this section:

- AUTHENTICATE Function
- ChangePassword Function

AUTHENTICATE Function

The AUTHENTICATE function authenticates a PL/SQL caller. Authentication is achieved by checking the supplied user name, password, and domain name against the local e-mail server. Authentication fails if no such user can be found. This function must be called before any other API functions in an IM_API session.

Syntax for the AUTHENTICATE Function

```
FUNCTION AUTHENTICATE(
    p_user    IN  VARCHAR2,
    password  IN  VARCHAR2,
    domain    IN  VARCHAR2   DEFAULT NULL)
RETURN INTEGER;
```

Input for the AUTHENTICATE Function

Table 2–9

Input	Description
p_user	This parameter contains the e-mail server user name
password	This parameter contains the unencrypted password for the user
domain	This parameter contains the fully qualified domain name that the user belongs to. If the user name is unique across all domains within the server, this parameter can be passed as NULL.

Returns of the AUTHENTICATE Function

Table 2–10

Returned Message	Description
IMAPI_OK	Success
IMAPI_ERR_NOUSER	User not found

Table 2–10

Returned Message	Description
IMAPI_ERR_DUPUSER	Ambiguous user name. This means domain name is NULL while the user name is not unique across the server.
IMAPI_ERR_FAUTH	Authentication failed. This means the password does not match the one in the server.

ChangePassword Function

This function changes the password on behalf of an authenticated user. This function requires AUTHENTICATE function being called already in the user session.

Syntax for the ChangePassword Function

```
FUNCTION ChangePassword(
    newpass  IN  VARCHAR2)
RETURN INTEGER;
```

Input for the ChangePassword Function

Table 2–11

Input	Description
newpass	New unencrypted password

Returns of the ChangePassword Function

Table 2–12

Returned Message	Description
IMAPI_OK	Success
IMAPI_ERR_FAUTH	Not authenticated. This means that the AUTHENTICATE function has not been called.

Logging Functions

The following functions are described in this section:

- EnableLogging Function
- DisableLogging Function

EnableLogging Function

This function starts logging debug messages from within IM_API to a specified location.

Syntax for the EnableLogging Function

```
FUNCTION EnableLogging(
   logdir    IN  VARCHAR2,
   logfile   IN  VARCHAR2,
   loglevel  IN  NUMBER)
RETURN INTEGER;
```

Input for the EnableLogging Function

Table 2–13

Input	Description
logdir	Directory on the server where log file should be created. This directory must be an absolute file system path name that's present in the list of directories where logging is allowed from PL/SQL. The directory list can be set using server parameter UTL_FILE_DIR in the init<*SID*>.ora file.
logfile	File name to be created for this log session
loglevel	A value between 1 and 5 to specify the detail level of the logs, with 5 being the most detailed level and 1 being the least detailed level.

Returns of the EnableLogging Function

Table 2–14

Returned Message	Description
IMAPI_OK	Success

DisableLogging Function

This function stops logging debug messages in the current session.

Syntax for the DisableLogging Function

```
FUNCTION DisableLogging RETURN INTEGER;
```

Returns of the DisableLogging Function

Table 2–15

Returned Message	Description
IMAPI_OK	Success

Folder Management Functions

The following functions are described in this section:

- CreateFolder Function
- DeleteFolder Function
- RenameFolder Function
- GetFolderExp Function
- SetFolderExp Function

CreateFolder Function

This function creates a folder using a specified name.

Syntax for the CreateFolder Function

```
FUNCTION CreateFolder(
   folder  IN  VARCHAR2)
RETURN INTEGER;
```

Input for the CreateFolder Function

Table 2–16

Input	Description
folder	The absolute path name of the folder that is to be created. The path name uses '/' as a delimiter, and must start with a leading '/' as well. Folder names are case sensitive with spaces allowed in the name. This function also creates the whole hierarchy of folders if intermediate folders specified in the path name are not present. For example, if the string '/First Level/Second Level/Third Level' is passed and none of the three folders are created, then this function creates all three of them in the exact hierarchical order.

Returns of the CreateFolder Function

Table 2–17

Returned Message	Description
IMAPI_OK	Success
IMAPI_ERR_FAUTH	Not authenticated. This means that the AUTHENTICATE function has not been called.

DeleteFolder Function

This function deletes a specified folder.

Syntax for the DeleteFolder Function

```
FUNCTION DeleteFolder(
   folder  IN  VARCHAR2)
RETURN INTEGER;
```

Input for the DeleteFolder Function

Table 2–18

Input	Description
folder	The absolute path name of the folder to be deleted. This function deletes only the folder in the deepest level specified. By deleting a folder, all the messages and subfolders are deleted.

Returns of the DeleteFolder Function

Table 2–19

Returned Message	Description
IMAPI_OK	Success
IMAPI_ERR_FAUTH	Not authenticated. This means that the AUTHENTICATE function has not been called.
IMAPI_ERR_NOFOLDER	No such folder. This means folder path passed in is invalid.

RenameFolder Function

This function renames a specified folder.

Syntax

```
FUNCTION RenameFolder(
   folder   IN  VARCHAR2,
   newname  IN  VARCHAR2)
RETURN INTEGER;
```

Input for the RenameFolder Function

Table 2–20

Input	Description
folder	The absolute path name of the folder to be renamed
newname	The new folder name without leading parent folder paths in front. For example if you want to rename a folder 'Sent' to 'Sent Items' under parent folder 'Private', call this function using the following syntax: `RenameFolder('/Private/Sent', 'Sent Items');`

Returns of the RenameFolder Function

Table 2–21

Returned Message	Description
IMAPI_OK	Success
IMAPI_ERR_FAUTH	Not authenticated. This means that the AUTHENTICATE function has not been called.
IMAPI_ERR_NOFOLDER	No such folder. This means folder path passed in is invalid.

GetFolderExp Function

This function retrieves the expiration property of a folder.

Syntax for the GetFolderExp Function

```
FUNCTION GetFolderExp(
   folder  IN   VARCHAR2,
   expiry  OUT  INTEGER)
```

```
RETURN INTEGER;
```

Input for the GetFolderExp Function

Table 2–22

Input	Description
folder	The absolute path name of the folder to be queried

Output of the GetFolderExp Function

Table 2–23

Output	Description
expiry	The number of days before messages in this folder expires. A value of NULL or 0 means messages in this folder never expires. An expired message will be removed from the system automatically after the expiration date. The expiration date is set to the date when the message is moved or delivered into a folder plus the expiry date of the folder.

Returns of the GetFolderExp Function

Table 2–24

Returned Message	Description
IMAPI_OK	Success
IMAPI_ERR_FAUTH	Not authenticated. This means that the AUTHENTICATE function has not been called.
IMAPI_ERR_NOFOLDER	No such folder. This means folder path passed in is invalid.

SetFolderExp Function

This function sets the expiration property of a folder.

Syntax for the SetFolderExp Function

```
FUNCTION SetFolderExp(
   folder  IN  VARCHAR2,
   expiry  IN  INTEGER)
RETURN INTEGER;
```

Input for the SetFolderExp Function

Table 2–25

Input	Description
folder	The absolute path name of the folder to be queried
expiry	The number of days before messages in this folder expires

Returns of the SetFolderExp Function

Table 2–26

Returned Message	Description
IMAPI_OK	Success
IMAPI_ERR_FAUTH	Not authenticated. This means that the AUTHENTICATE function has not been called.
IMAPI_ERR_NOFOLDER	No such folder. This means folder path passed in is invalid.

Message Generation Functions

The following functions are described in this section:

- BlindCopyTo Function
- ForwardTo Function
- ForwardWithTemplate Function
- ReplyTo Function
- ReplyWithTemplate Function
- SendSimpleMessage Function
- ListTemplates Function

BlindCopyTo Function

This function generates a blind carbon copy (BCC) of the message to a recipient.

Syntax for the BlindCopyTo Function

```
FUNCTION BlindCopyTo(
   message   IN  INTEGER,
```

```
   to_recip  IN  VARCHAR2)
RETURN INTEGER;
```

Input for the BlindCopyTo Function

Table 2–27

Input	Description
message	Message ID of the message to be blind carbon copied
to_recip	Recipient address that should received the blind carbon copy

Returns of the BlindCopyTo Function

Table 2–28

Returned Message	Description
IMAPI_OK	Success
IMAPI_ERR_FAUTH	Not authenticated. This means that the AUTHENTICATE function has not been called.
IMAPI_ERR_NOMSG	No such message. This means the message ID passed in is invalid.

ForwardTo Function

The ForwardTo function forwards a message to another recipient with additional simple comments.

Syntax for the ForwardTo Function

```
FUNCTION ForwardTo(
   msgid    IN  INTEGER,
   recip    IN  VARCHAR2,
   notes  IN  VARCHAR2,
   subject  IN  VARCHAR2   DEFAULT NULL,
   fromstr  IN  VARCHAR2   DEFAULT NULL,
   replyto  IN  VARCHAR2   DEFAULT NULL)
RETURN INTEGER;
```

Input for the ForwardTo Function

Table 2–29

Input	Description
msgid	Message ID of the message to be forwarded
recip	The recipient address that should received the forwarded message
notes	The forward comment that accompanies the original message
subject	Message subject
fromstr	From string of the forwarded message
replyto	The Reply-to header of the forwarded message

ForwardWithTemplate Function

The ForwardWithTemplate function forwards a message using an existing template for comments. For example, an auto-forward template can be attached to a forwarding message.

Syntax for ForwardWithTemplate Function

```
FUNCTION ForwardWithTemplate(
   msgid    IN  INTEGER,
   recip    IN  VARCHAR2,
   templid  IN  INTEGER)
RETURN INTEGER;
```

Input for ForwardWithTemplate Function

Table 2–30

Input	Description
msgid	Message ID of the message being replied to
recip	Recipient address that should receive the forwarded message
templid	Template ID of the template used to reply to the message

ReplyTo Function

The ReplyTo function replies to a message ID with additional simple comments.

Syntax for the ReplyTo Function

```
FUNCTION ReplyTo(
   msgid    IN  INTEGER,
   notes  IN  VARCHAR2,
   subject  IN  VARCHAR2   DEFAULT NULL,
   fromstr  IN  VARCHAR2   DEFAULT NULL,
   replyto  IN  VARCHAR2   DEFAULT NULL)
RETURN INTEGER;
```

Input for the ReplyTo Function

Table 2–31

Input	Description
msgid	Message ID of the message being replied to
notes	The forward comment that accompanies the original message
subject	Message subject
fromstr	From string of the forwarded message
replyto	Reply-to header of the forwarded message

ReplyWithTemplate Function

The `ReplyWithTemplate` function replies to a message using an existing template for comments. For example, a vacation reply template can be attached to a replying message.

Syntax for the ReplyWithTemplate Function

```
FUNCTION ReplyWithTemplate(
   msgid    IN  INTEGER,
   templid  IN  INTEGER)
RETURN INTEGER;
```

Input for the ReplyWithTemplate Function

msgid	Message ID of the message being replied to
templid	Template ID of the template used to reply to the message

SendSimpleMessage Function

The SendSimpleMessage function sends a simple text message with no attachments.

Syntax for the SendSimpleMessage Function

```
FUNCTION SendSimpleMessage(
   recip     IN   VARCHAR2,
   comment   IN   VARCHAR2,
   subject   IN   VARCHAR2,
   fromstr   IN   VARCHAR2,
   replyto   IN   VARCHAR2)
RETURN INTEGER;
```

Input for the SendSimpleMessage Function

Table 2–32

Input	Description
recip	Recipient address for the message
comment	The forward comment that accompanies the original message
subject	Message subject
fromstr	From string of the forwarded message
replyto	Reply-to header of the forwarded message

ListTemplates Function

This function lists all private templates for the user into a PL/SQL array. Every template in the array is an object containing its template ID and its message subject. This function can be used to retrieve template IDs to call the ReplyWithTemplate or ForwardWithTemplate functions.

Syntax for the ListTemplates Function

```
FUNCTION ListTemplates(
   p_templates  IN OUT  api_template_c)
RETURN INTEGER;
```

Output of the ListTemplates Function

Table 2–33

Output	Description
p_templates	The array of template descriptors, which themselves are of type api_template_t, a PL/SQL object type declared as follows: CREATE OR replace TYPE api_template_t AS object (msg_id NUMBER, subject VARCHAR2(240)); CREATE OR replace TYPE api_template_c AS TABLE OF api_template_t;

Returns of the ListTemplates Function

Table 2–34

Returned Message	Description
IMAPI_OK	Success
IMAPI_ERR_FAUTH	Not authenticated. This means the AUTHENTICATE function has not been called.

Checking New Mail Functions

The following functions are described in this section:

- HasNewMail Function
- GetNewMail Function

HasNewMail Function

This function returns whether or not there is new e-mail for a user. This function can be used with no authentication if a valid user name and domain name is supplied.

Syntax for the HasNewMail Function

```
FUNCTION HasNewMail(
   p_user    IN  VARCHAR2   DEFAULT NULL,
   p_domain  IN  VARCHAR2   DEFAULT NULL)
RETURN BOOLEAN;
```

Input for the HasNewMail Function

Table 2–35

Input	Description
p_user	User name whose Inbox is queried for new e-mail existence. If this parameter is omitted, then the server uses the currently authenticated user.
p_domain	Fully qualified domain name for the user

Returns of the HasNewMail Function

Table 2–36

Returned Message	Description
TRUE	There is new e-mail for the user defined in p_user
FALSE	There is no new e-mail for the user defined in p_user

GetNewMail Function

This function returns an array of message IDs of all of new mails.

Syntax for the GetNewMail Function

```
FUNCTION GetNewMail(
   messages  OUT  msg_table)
RETURN INTEGER;
```

Output of the GetNewMail Function

Table 2–37

Output	Description
messages	The list of message IDs of all the new e-mails. The type msg_table is a PL/SQL index-by table declared as follows: `TYPE msg_table IS TABLE OF INTEGER INDEX BY BINARY_INTEGER;`

Returns of the GetNewMail Function

Table 2–38

Returned Message	Description
IMAPI_OK	Success
IMAPI_ERR_FAUTH	Not authenticated. This means the AUTHENTICATE function has not been called.

Message Navigation Functions

The following functions are described in this section:

- ListFolders Function
- OpenFolder Function (Array Interface)
- OpenFolder Function (Iterative Interface)
- GetNextMessage Function
- CloseFolder Function

Message navigation can be achieved by two different ways. You can either retrieve all message IDs into a PL/SQL array and then process the messages based on that array, or open a folder and repeatedly retrieve messages until the GetNextMessage function returns a value indicating no more messages are present in the folder.

ListFolders Function

This function lists all private folders for the user into a PL/SQL array. Every folder descriptor in the array is an object containing its folder ID and its hierarchical folder name.

Syntax for the ListFolders Function

```
FUNCTION ListFolders(
   p_folders  IN OUT  api_folder_c)
RETURN INTEGER;
```

Output of the ListFolders Function

Table 2–39

Output	Description
p_folders	The array of folder descriptors, which themselves are of type api_folder_t, a PL/SQL object type declared as follows: CREATE OR replace TYPE api_folder_t AS object folder_id NUMBER, folder_path VARCHAR2(240)); CREATE OR replace TYPE api_folder_c AS TABLE OF api_folder_t;

Returns of the ListFolders Function

Table 2–40

Returned Message	Description
IMAPI_OK	Success
IMAPI_ERR_FAUTH	Not authenticated. This means the AUTHENTICATE function has not been called.

OpenFolder Function (Array Interface)

This function retrieves all message IDs inside a folder in an PL/SQL index-by table.

Syntax for the OpenFolder Function (Array Interface)

```
FUNCTION OpenFolder(
   folder    IN   VARCHAR2,
   messages  OUT  msg_table)
RETURN INTEGER;
```

Input for the OpenFolder Function (Array Interface)

Table 2–41

Input	Description
folder	The absolute path name of the folder to be opened.

Output of the OpenFolder Function (Array Interface)

Table 2–42

Output	Description
messages	The list of message IDs inside the folder. The type msg_table is a PL/SQL index-by table declared as follows: TYPE msg_table IS TABLE OF INTEGER INDEX BY BINARY_INTEGER;

Returns of the OpenFolder Function (Array Interface)

Table 2–43

Returned Message	Description
IMAPI_OK	Success
IMAPI_ERR_FAUTH	Not authenticated. This means the AUTHENTICATE function has not been called.
IMAPI_ERR_NOFOLDER	No such folder. This means folder path passed in is invalid.

OpenFolder Function (Iterative Interface)

The OpenFolder function opens a named folder and returns the number of messages in that folder, the folder ID, the folder size in terms of total size of the messages residing in the folder, and folder type.

Syntax for the OpenFolder Function (Iterative Interface)

```
FUNCTION OpenFolder(
   fname   IN   VARCHAR2,
   count   OUT  INTEGER,
   fid     OUT  INTEGER,
   fsize   OUT  INTEGER,
   ftype   OUT  INTEGER)
RETURN INTEGER;
```

Input for the OpenFolder Function (Iterative Interface)

Table 2–44

Input	Description
fname	Name of the folder to be opened

Output of the OpenFolder Function (Iterative Interface)

Table 2–45

Output	Description
count	Number of messages in the folder
fid	Folder ID of the opened folder
fsize	Total size of the folder
ftype	Folder type which takes one of the following values: 0 (regular private folders) 1 (inbox) 2 (outbox) 3 (wastebasket) 4 (hidden folders) 5 (draft folders) 6 (template folders) 9 (shared folders) 10 (public folders)

GetNextMessage Function

The GetNextMessage function gets the next message in the currently opened folder. The API internally keeps a open cursor on the folder across iterative calls to this function.

Syntax for the GetNextMessage Function

```
FUNCTION GetNextMessage(
   msgid  OUT  INTEGER)
RETURN INTEGER;
```

Output of the GetNextMessage Function

Table 2–46

Output	Description
`msgid`	Message ID of the next message in the opened folder

CloseFolder Function

The `CloseFolder` function closes the currently opened folder. There can only be one opened folder at a time. This call enables you to close an open folder without finishing retrieving all the messages from it.

Syntax for the CloseFolder Function

```
FUNCTION CloseFolder RETURN INTEGER;
```

Message Foldering Functions

The following functions are described in this section:

- CopyToFolder Function (Hierarchical Folder Interface)
- MoveToFolder Function (Hierarchical Folder Interface)
- DeleteFrom Function (Hierarchical Folder Interface)
- CopyMessage Function (Non-Hierarchical Folder Interface)
- MoveMessage Function (Non-Hierarchical Folder Interface)
- DeleteMessage Function (Non-Hierarchical Folder Interface)

Foldering operations include move, copy, and delete messages. Deleting a message is equivalent to moving a message from its original folder to the user's dedicated Wastebasket. For copy, move and delete operations, you can specify folders in either the hierarchical format or the non-hierarchical format using two different sets of functions. The former is recommended since it provides greater flexibility in organizing folders. The latter is limited to system with no folder hierarchy, or only the top-level folders in a system with hierarchical folders. Another key difference is that the first set of functions uses case-sensitive folder names and the latter uses case-insensitive folder names. For example, to specify the Inbox folder, its hierarchical name is /Inbox and the non-hierarchical name is Inbox.

CopyToFolder Function (Hierarchical Folder Interface)

This function copies a message from a source folder to a target folder.

Syntax for the CopyToFolder Function

```
FUNCTION CopyToFolder(
  message  IN  INTEGER,
  source   IN  VARCHAR2,
  target   IN  VARCHAR2)
RETURN INTEGER;
```

Input for the CopyToFolder Function

Table 2–47

Input	Description
message	Message ID of the message to be copied
source	The absolute path name of the folder from which the message is to be copied
target	The absolute path name of the folder to which the message is to be copied

Returns of the CopyToFolder Function

Table 2–48

Returned Message	Description
IMAPI_OK	Success
IMAPI_ERR_FAUTH	Not authenticated. This means the AUTHENTICATE function has not been called.
IMAPI_ERR_NOFOLDER	No such folder. This means folder path passed in is invalid.
IMAPI_ERR_NOMSG	No such message. This means the message ID passed in is invalid.

MoveToFolder Function (Hierarchical Folder Interface)

This function moves a message from a source folder to a target folder.

Syntax for the MoveToFolder Function

```
FUNCTION MoveToFolder(
   message  IN  INTEGER,
   source   IN  VARCHAR2,
   target   IN  VARCHAR2)
RETURN INTEGER;
```

Input for the MoveToFolder Function

Table 2–49

Input	Description
message	Message ID of the message to be moved
source	The absolute path name of the folder from which the message is to be moved
target	The absolute path name of the folder to which the message is to be moved

Returns of the MoveToFolder Function

Table 2–50

Returned Message	Description
IMAPI_OK	Success
IMAPI_ERR_FAUTH	Not authenticated. This means the AUTHENTICATE function has not been called.
IMAPI_ERR_NOFOLDER	No such folder. This means folder path passed in is invalid.
IMAPI_ERR_NOMSG	No such message. This means the message ID passed in is invalid.

DeleteFrom Function (Hierarchical Folder Interface)

This function deletes a message from a source folder.

Syntax for the DeleteFrom Function

```
FUNCTION DeleteFrom(
   message  IN  INTEGER,
   source   IN  VARCHAR2)
RETURN INTEGER;
```

Input for the DeleteFrom Function

Table 2–51

Input	Description
message	Message ID of the message to be deleted
source	The absolute path name of the folder from which the message is to be deleted

Returns of the DeleteFrom Function

Table 2–52

Returned Message	Description
IMAPI_OK	Success
IMAPI_ERR_FAUTH	Not authenticated. This means the AUTHENTICATE function has not been called.
IMAPI_ERR_NOFOLDER	No such folder. This means folder path passed in is invalid.
IMAPI_ERR_NOMSG	No such message. This means the message ID passed in is invalid.

CopyMessage Function (Non-Hierarchical Folder Interface)

The CopyMessage function copies a message from one folder to another folder.

Syntax for the CopyMessage Function

```
FUNCTION CopyMessage(
   msgid  IN  INTEGER,
   fromfld  IN  VARCHAR2,
   tofld    IN  VARCHAR2)
RETURN INTEGER;
```

Input for the CopyMessage Function

Table 2–53

Input	Description
msgid	Message ID of the message to be copied

Table 2–53

Input	Description
fromfld	Non-hierarchical source folder name of the message to be copied
tofld	Non-hierarchical destination folder name of the message to be copied

MoveMessage Function (Non-Hierarchical Folder Interface)

The MoveMessage function moves a message from one folder to another folder.

Syntax for the MoveMessage Function

```
FUNCTION MoveMessage(
   msgid    IN  INTEGER,
   fromfld  IN  VARCHAR2,
   tofld    IN  VARCHAR2)
RETURN INTEGER;
```

Input for the MoveMessage Function

Table 2–54

Input	Description
msgid	Message ID of the message to be moved
fromfld	Non-hierarchical source folder name of the message to be moved
tofld	Non-hierarchical destination folder name of the message to be moved

DeleteMessage Function (Non-Hierarchical Folder Interface)

The DeleteMessage function deletes a message from one folder.

Syntax for the DeleteMessage Function

```
FUNCTION DeleteMessage(
   msgid    IN  INTEGER,
   fromfld  IN  VARCHAR2)
RETURN INTEGER;
```

Input for the DeleteMessage Function

Table 2–55

Input	Description
msgid	Message ID of the message to be deleted
fromfld	Non-hierarchical source folder name of the message to be deleted

Message Property Functions

The following functions are described in this section:

- GetMessageProps Function (Bundled Interface)
- SetMessageProps Function (Bundled Interface)
- GetMessageProp Function (Non-Bundled Interface)
- SetMessageProp Function (Non-Bundled Interface)

You can retrieve and save properties of a message instance either in one bundled function or do it one at a time for each property. The former is recommended if you need to manage more than one properties for a message. Another difference is that the bundled interface use hierarchical folder names and the non-bundled interfaces uses non-hierarchical folder names.

GetMessageProps Function (Bundled Interface)

This function retrieves UID, priority, received date, expiration date, and read status for a message in a folder.

Syntax for the GetMessage Props Function (Bundled Interface)

```
FUNCTION GetMessageProps(
   message        IN   INTEGER,
   folder         IN   VARCHAR2,
   muid            OUT  INTEGER,
   priority       OUT  INTEGER,
   received_date  OUT  DATE,
   expiration     OUT  DATE)
   read           OUT  INTEGER)
RETURN INTEGER;
```

Input for the GetMessage Props Function (Bundled Interface)

Table 2–56

Input	Description
`message`	Message ID of the message to be queried
`folder`	The absolute path name of the folder from which the message resides

Output of the GetMessage Props Function (Bundled Interface)

Table 2–57

Output	Description
`uid`	Message UID used in IMAP, POP and JMA to uniquely identify a message in a folder
`priority`	Priority of the message; a value of 50, 0, or -50 indicates the priority being high, normal, or low, respectively
`received_date`	The date and time of when the message was received
`expiration`	The date when the message will expire, if any
`read`	Whether the message is marked "read" or not. `0` (the message is not read) `1` (the message is read)

Returns of the GetMessage Props Function (Bundled Interface)

Table 2–58

Returned Message	Description
`IMAPI_OK`	Success
`IMAPI_ERR_FAUTH`	Not authenticated. This means the `AUTHENTICATE` function has not been called.
`IMAPI_ERR_NOFOLDER`	No such folder. This means folder path passed in is invalid.
`IMAPI_ERR_NOMSG`	No such message. This means the message ID passed in is invalid.

SetMessageProps Function (Bundled Interface)

This function sets priority, expiration date and read status for a message in a folder.

Syntax for the SetMessageProps Function (Bundled Interface)

```
FUNCTION SetMessageProps(
   message     IN  INTEGER,
   folder      IN  VARCHAR2,
   priority    IN  INTEGER,
   expiration  IN  DATE,
   read        IN  INTEGER   DEFAULT -1)
RETURN INTEGER;
```

Input for the SetMessageProps Function (Bundled Interface)

Table 2–59

Input	Description
`message`	Message ID of the message to be queried
`folder`	The absolute path name of the folder from which the message resides
`priority`	Priority of the message; a value of 50, 0, or -50 indicates the priority being high, normal, or low, respectively
`expiration`	The date when the message will expire, if any
`read`	Whether the message is marked "read" or not. *Valid values:* `0` (the message is not read) `1` (the message is read)

Returns of the SetMessageProps Function (Bundled Interface)

Table 2–60

Returned Message	Description
`IMAPI_OK`	Success
`IMAPI_ERR_FAUTH`	Not authenticated. This means the `AUTHENTICATE` function has not been called.
`IMAPI_ERR_NOFOLDER`	No such folder. This means folder path passed in is invalid.

Table 2–60

Returned Message	Description
IMAPI_ERR_NOMSG	No such message. This means the message ID passed in is invalid.

GetMessageProp Function (Non-Bundled Interface)

The GetMessageProp function retrieves dynamic properties associated with a message instance; examples include the message priority and message expiration date. Static properties of the message are usually called messages headers, which are read-only properties. The caller can change dynamic properties.

Syntax for GetMessageProp Function (Non-Bundled Interface)

```
FUNCTION GetMessageProp(
   msgid     IN   INTEGER,
   fname     IN   VARCHAR2,
   propflag  IN   INTEGER,
   valuen    OUT  INTEGER,
   valuev    OUT  VARCHAR2,
   valued    OUT  DATE)
RETURN INTEGER;
```

Input for GetMessageProp Function (Non-Bundled Interface)

Table 2–61

Input	Description
msgid	Message ID of the message to be retrieved
fname	Name of the folder containing the message to be retrieved
propflag	The following five properties can be used: IM_API_PROP_EXPR (expiration date) IM_API_PROP_PRIO (priority) IM_API_PROP_READ (read status) IM_API_PROP_RECD (received date) IM_API_PROP_MUID (message UID)

Output for GetMessageProp Function (Non-Bundled Interface)

Table 2–62

Output	Description
valuen	Contains the numerical value of the propflag parameter if propflag is set to IM_API_PROP_PRIO, IM_API_PROP_READ, or IM_API_PROP_MUID.
valuev	Contains VARCHAR2 properties if present. Currently not used.
valued	Contains the date value of the propflag parameter if propflag is set to IM_API_PROP_EXPR or IM_API_PROP_RECD.

SetMessageProp Function (Non-Bundled Interface)

The SetMessageProp function sets dynamic properties associated with a message instance.

Syntax for the SetMessageProp Function (Non-Bundled Interface)

```
FUNCTION SetMessageProp(
   msgid     IN  INTEGER,
   fname     IN  VARCHAR2,
   propflag  IN  INTEGER,
   valuen    IN  INTEGER,
   valuev    IN  VARCHAR2,
   valued    IN  DATE)
RETURN INTEGER;
```

Input for the SetMessageProp Function (Non-Bundled Interface)

Table 2–63

Input	Description
msgid	Message ID of the message to be modified
fname	Name of the folder containing the message to be modified
propflag	The following five properties can be used: IM_API_PROP_EXPR (expiration date) IM_API_PROP_PRIO (priority) IM_API_PROP_READ (read status)

Table 2–63

Input	Description
valuen	Contains the numerical value of the propflag parameter if propflag is set to IM_API_PROP_PRIO or IM_API_PROP_READ.
valuev	Contains VARCHAR2 properties if present. Currently not used.
valued	Contains the date value of the propflag parameter if propflag is set to IM_API_PROP_EXPR.

Message Parsing Functions

The following functions are described in this section:

- GetPartList Function
- GetMessageHdrs Function
- GetExtendedHdrs Function
- GetMessageBody Function
- GetAttachmentBody Function
- GetAttachmentData Function
- GetNextAttachment Function
- OpenMessageAttachment Function
- GetInclusionID Function

You can retrieve a list of message attachments or MIME body parts either by using the new array interface GetPartList, or by using the iterative method to fetch one attachment ID at a time. The new function is recommended not only because it retrieves the attachment IDs and descriptors in one array, but also, because the function GetPartList is MIME knowledgeable, it will only return a list of unique body parts within a MIME message and discard duplicate entries inside a multipart/alternative block. You may still want to use the old function if you want to retrieve all body parts regardless of their MIME context, or if you want to retrieve "poorer" alternatives from a multipart/alternative MIME block. The new function GetPartList returns only the "richest" alternative if possible. The *richness* of a body part is defined as the relative detail contained in a particular presentation format for that body part. For example, a body part in HTML format is considered to be richer than the same body part in plain text format.

GetPartList Function

This function retrieves a list of unique body part descriptors from a MIME message. The list of descriptors is saved in a PL/SQL index-by table of PL/SQL records. The descriptor can then be used to retrieve the content the body part. If there's a "multipart/alternative" MIME block inside the message, only the best alternative is represented in the list of body part descriptors.

Syntax for the GetPartList Function

```
FUNCTION GetPartList(
   message      IN   INTEGER,
   attachments  OUT  att_table)
RETURN INTEGER;
```

Input for the GetPartList Function

Table 2–64

Input	Description
message	Message ID of the message to be queried

Output of the GetPartList Function

Table 2–65

Output	Description

attachments	The array of attachment descriptors. Every descriptor contains body part number, MIME content-type, binary or textual nature of the part, the part size and name if present. The PL/SQL array type att_table is declared as follows:
	`TYPE att_record IS RECORD(`
	`  part_number  INTEGER,`
	`  content_type  VARCHAR2(240),`
	`  is_binary  VARCHAR2(1),`
	`  att_size  INTEGER,`
	`  att_name  VARCHAR2(240));`
	`TYPE att_table IS TABLE OF att_record INDEX BY BINARY_INTEGER;`

Returns of the GetPartList Function

Table 2–66

Returned Message	Description
IMAPI_OK	Success
IMAPI_ERR_FAUTH	Not authenticated. This means the AUTHENTICATE function has not been called.
IMAPI_ERR_NOMSG	No such message. This means the message ID passed in is invalid.

GetMessageHdrs Function

The GetMessageHdrs function retrieves the message headers.

Syntax for the GetMessageHdrs Function

```
FUNCTION GetMessageHdrs(
   msgid      IN   INTEGER,
   subject    OUT  VARCHAR2,
   sender     OUT  VARCHAR2,
   to_recip   OUT  VARCHAR2,
   cc_recip   OUT  VARCHAR2,
   from_str   OUT  VARCHAR2,
   sent_date  OUT  DATE,
   reply_to   OUT  VARCHAR2,
   msg_size   OUT  INTEGER)
RETURN INTEGER;
```

Input for the GetMessageHdrs Function

Table 2–67

Input	Description
msgid	Message ID of the message to be retrieved

Output of the GetMessageHdrs Function

Table 2–68

Output	Description
subject	Message subject

Table 2–68

Output	Description
sender	Sender address of the message
to_recip	Recipient list in the To header
cc_recip	Recipient list in the CC header
from_str	From string of the message
sent_date	Date and time when the message was sent
reply_to	Reply-to header of the message
msg_size	Size of the message

GetExtendedHdrs Function

This function retrieves extended message headers from a message. Extended headers are message headers that are not part of the standard headers returned by function GetMessageHdrs. The extended headers retrieved are passed out in two arrays, one storing all the header names and one storing all the header values.

Syntax for the GetExtendedHdrs Function

```
FUNCTION GetExtendedHdrs(
   message    IN   INTEGER,
   hdrnames   OUT  name_table,
   hdrvalues  OUT  value_table)
RETURN INTEGER;
```

Input for the GetExtendedHdrs Function

Table 2–69

Input	Description
message	Message ID of the message to be queried

Output of the GetExtendedHdrs Function

Table 2–70

Output	Description
hdrnames	The array of header names. The PL/SQL array type name_table is declared as follows: TYPE name_table IS TABLE OF VARCHAR2(30) INDEX BY BINARY_INTEGER;
hdrvalues	The array of header values. The PL/SQL array type value_table is declared as follows: TYPE value_table IS TABLE OF VARCHAR2(240) INDEX BY BINARY_INTEGER;

Returns of the GetExtendedHdrs Function

Table 2–71

Returned Message	Description
IMAPI_OK	Success
IMAPI_ERR_FAUTH	Not authenticated. This means the AUTHENTICATE function has not been called.
IMAPI_ERR_NOMSG	No such message. This means the message ID passed in is invalid.

GetMessageBody Function

The GetMessageBody function retrieves the main message body. This function should be called in a loop to retrieve the e-mail message body one piece at a time. The parameter restart can be used to indicate the start of a new message retrieval. The return value can be used to determine whether the message has ended.

Syntax for the GetMessageBody Function

```
FUNCTION GetMessageBody(
   msgid    IN      INTEGER,
   restart  IN      BOOLEAN,
   buflen   IN OUT  INTEGER,
   msg_buf  OUT     VARCHAR2)
RETURN INTEGER;
```

Input for the GetMessageBody Function

Table 2–72

Input	Description
msgid	Message ID of the message to be retrieved
restart	Indicates whether this is the start of a new message
buflen	Length of the buffer passed in

Output of the GetMessageBody Function

Table 2–73

Output	Description
buflen	The actual length returned in msg_buf
msg_buf	Contains the message body up to the value of buflen, or the whole message body if it is smaller than buflen

GetAttachmentBody Function

The GetAttachmentBody function gets the attachment content if it is a textual attachment. It should be called in the same fashion as GetMessageBody function. The parameter part corresponds to the attachment ID returned of the function GetNextAttachment.

Syntax for the GetAttachmentBody Function

```
FUNCTION GetAttachmentBody(
   msgid    IN      INTEGER,
   part     IN      INTEGER,
   restart  IN      BOOLEAN,
   buflen   IN OUT  INTEGER,
   msg_buf  OUT     VARCHAR2)
RETURN INTEGER;
```

Input for the GetAttachmentBody Function

Table 2–74

Input	Description
msgid	Message ID of the message to be retrieved

Table 2–74

Input	Description
part	Attachment ID of the attachment to be retrieved
restart	Indicates whether this is the start of a new message
buflen	Length of the buffer passed in

Output of the GetAttachmentBody Function

Table 2–75

Output	Description
buflen	The actual length returned in msg_buf
msg_buf	Contains the message body up to the value of buflen, or the whole message body if it is smaller than buflen

GetAttachmentData Function

The GetAttachmentData function retrieves the attachment content if it is a binary attachment. It should be called in the same fashion as GetMessageBody function. The parameter part corresponds to the attachment ID returned of the function GetNextAttachment.

Syntax for the GetAttachmentData Function

```
FUNCTION GetAttachmentData(
   msgid    IN      INTEGER,
   part     IN      INTEGER,
   restart  IN      BOOLEAN,
   buflen   IN OUT  INTEGER,
   msg_buf  OUT     RAW)
RETURN INTEGER;
```

Input for the GetAttachmentData Function

Table 2–76

Input	Description
msgid	Message ID of the message to be retrieved
part	Attachment ID of the attachment to be retrieved

Table 2–76

Input	Description
restart	Indicates whether this is the start of a new message
buflen	Length of the buffer passed in

Output for the GetAttachmentData Function

Table 2–77

Output	Description
buflen	The actual length returned in msg_buf
msg_buf	The raw buffer holding the attachment content

GetNextAttachment Function

The GetNextAttachment function gets the next attachment ID within the same message. The attachment ID can be used in conjunction with the message ID to retrieve the attachment content.

Syntax for the GetNextAttachment Function

```
FUNCTION GetNextAttachment(
   msgid     IN   INTEGER,
   att_id    OUT  INTEGER,
   att_type  OUT  VARCHAR)
RETURN INTEGER;
```

Input for the GetNextAttachment Function

Table 2–78

Input	Description
msgid	Message ID of the message to be retrieved

Output of the GetNextAttachment Function

Table 2–79

Output	Description
att_id	An order number within the message to represent a unique attachment
att_type	Indicate whether this attachment is binary or textual. Y (binary) N (text)

OpenMessageAttachment Function

The OpenMessageAttachment function opens a list of attachments for a message. This function must be called in conjunction with the GetNextAttachment function to iterate through the list of attachments.

Syntax for the OpenMessageAttachment Function

```
FUNCTION OpenMessageAttachment(
   msgid  IN  INTEGER)
RETURN INTEGER;
```

Input for the OpenMessageAttachment Function

Table 2–80

Input	Description
msgid	Message ID of the message to be retrieved

GetInclusionID Function

The GetInclusionID function gets the included message ID in the current message.

Syntax for the GetInclusionID Function

```
FUNCTION GetInclusionID(
   msgid    IN   INTEGER,
   inc_msg  OUT  INTEGER)
RETURN INTEGER;
```

Input for the GetInclusionID Function

Table 2–81

Input	Description
msgid	Parent message

Output for the GetInclusionID Function

Table 2–82

Output	Description
inc_msg	Included message

IM_API Usage Sample Script

The following example uses IM_API to retrieve all of the messages and their attachments or included/forwarded messages in a user's Inbox folder. It then uses the DBMS_OUTPUT package supplied by the Oracle RDBMS to display all of the contents retrieved to the SQL*PLUS terminal.

Example 1: Retrieving Message Contents

The following example uses IM_API to retrieve all of the messages and their attachments or included/forwarded messages in a user's Inbox folder. It then uses the DBMS_OUTPUT package supplied by the Oracle RDBMS to display all of the contents retrieved to the SQL*PLUS terminal. It also forwards every message to a fixed address, generates an auto-reply, a notification, and finally deletes every message from Inbox. Before running the example, modify the constant declaration section to make sure all the values are valid.

This example can be accessed in the following directory:

```
$ORACLE_HOME/office/admin/samples/imapi_sample1.sql
```

```
DECLARE
   -- sample data
   c_username CONSTANT VARCHAR2(30) := 'scott';
   c_domain CONSTANT VARCHAR2(80) := 'acme.com';
   c_password CONSTANT VARCHAR2(80) := 'tiger';
   c_copyfolder CONSTANT VARCHAR2(30) := 'SENT';
   c_fwdrecip CONSTANT VARCHAR2(30) := 'tom@acme.com';
   c_fwdsubject CONSTANT VARCHAR2(30) := 'Forwarded mail';
```

```
   c_fwdbody CONSTANT VARCHAR2(80) := 'The following is forwarded from
     oracle email Server';
   c_resubject CONSTANT VARCHAR2(30) := 'Replied mail';
   c_rebody CONSTANT VARCHAR2(80) := 'The following is received by
     oracle email server';

   -- local variables
   restart BOOLEAN := true;
   folder_count INTEGER;
   folder_id INTEGER;
   folder_size INTEGER;
   folder_type INTEGER;
   msg_id INTEGER;
   top_id INTEGER;
   att_id INTEGER;
   inc_id INTEGER := 0;
   att_type CHAR;
   msg_buf VARCHAR2(240);
   raw_buf RAW(240);
   buf_len INTEGER;
   status INTEGER;
   subject VARCHAR2(240);
   from_str VARCHAR2(240);
   cc_str VARCHAR2(240);
   to_str VARCHAR2(240);
   replyto VARCHAR2(240);
   sender VARCHAR2(240);
   dummy VARCHAR2(240);
   sent_date DATE;
   msg_size INTEGER;
   expr_date DATE;
   prio INTEGER;
BEGIN
   -- Login first
   status := im_api.authenticate(c_username, c_domain, c_password);
   dbms_output.put_line('Authentication Status    : ' || status);

   -- Open INBOX
   status := im_api.openfolder('INBOX', folder_count, folder_id,
     folder_size, folder_type);
   dbms_output.put_line('Open Folder Status    : ' || status);

   LOOP
      -- Navigate INBOX
      status := im_api.getnextmessage(msg_id);
```

```
    IF status = im_api.imapi_err_nomsg THEN
 EXIT;
    END IF;

    -- retrieve message
    dbms_output.put_line('=== Begin message ' || msg_id || ' ===');

    -- Get and print the priority and expiration
    status := im_api.getmessageprop(msg_id, 'INBOX',
im_api.imapi_prop_expr, prio, dummy, expr_date);
    status := im_api.getmessageprop(msg_id, 'INBOX',
im_api.imapi_prop_prio, prio, dummy, expr_date);
    dbms_output.put_line('Expires : ' || expr_date);
    dbms_output.put_line('Priority: ' || prio);

    -- Parse the current mail message
    top_id := msg_id;
    LOOP
 -- Get and print the header
 status := im_api.getmessagehdrs(msg_id, subject, sender, to_str,
   cc_str, from_str, sent_date, replyto, msg_size);
 dbms_output.put_line('From    : ' || from_str);
 dbms_output.put_line('To      : ' || to_str);
 dbms_output.put_line('CC      : ' || cc_str);
 dbms_output.put_line('Sender  : ' || sender);
 dbms_output.put_line('Reply-To: ' || replyto);
 dbms_output.put_line('Subject : ' || subject);
 dbms_output.put_line('Date    : ' || sent_date);
 dbms_output.put_line('Size    : ' || msg_size);

 -- Get and print the main message body
 restart := true;
 dbms_output.put_line('=== Main message ===');
 LOOP
    buf_len := 240;
    status := im_api.getmessagebody(msg_id, restart, msg_buf,
           buf_len);
    IF status = im_api.imapi_nomoredata AND buf_len = 0 THEN
       EXIT;
    END IF;

    restart := false;
    dbms_output.put_line(msg_buf);
 END LOOP;
```

```
-- Open list of attachments
status := im_api.openmessageattachment(msg_id);
LOOP
   status := im_api.getnextattachment(msg_id, att_id, att_type);
   IF status = im_api.imapi_err_noatt THEN
      EXIT;
   END IF;

   dbms_output.put_line('=== Attachment ' || att_id || ' ===');
   restart := true;
   LOOP
      buf_len := 240;
      IF att_type = 'Y' OR att_type = 'y' THEN
  status := im_api.getattachmentdata(msg_id, att_id,
             restart, raw_buf, buf_len);
      ELSE
  status := im_api.getattachmentbody(msg_id, att_id,
             restart, msg_buf, buf_len);
      END IF;

      IF status = im_api.imapi_nomoredata AND buf_len = 0 THEN
  EXIT;
      END IF;

      restart := false;
      IF att_type = 'Y' OR att_type = 'y' THEN
  dbms_output.put_line(substrb(raw_buf, 1, 240));
  dbms_output.put_line(substrb(raw_buf, 241, 480));
      ELSE
  dbms_output.put_line(msg_buf);
      END IF;
   END LOOP;
END LOOP;

-- Get the included message
status := im_api.getinclusionid(msg_id, inc_id);
IF inc_id = 0 OR inc_id IS NULL THEN
   EXIT;
END IF;

msg_id := inc_id;
dbms_output.put_line('=== Included message ===');
    END LOOP;
    dbms_output.put_line('=== End message ===');
```

```
      -- forward and reply
      status := im_api.forwardto(msg_id, c_fwdrecip, c_fwdbody,
   c_fwdsubject, NULL, NULL);
      dbms_output.put_line('Forwarded with status ' || status);

      status := im_api.replyto(msg_id, c_rebody, c_resubject, NULL,
        NULL);
      dbms_output.put_line('Replied with status ' || status);

      -- copy and delete
      status := im_api.copymessage(msg_id, 'INBOX', c_copyfolder);
      dbms_output.put_line('Copied to folder ' || c_copyfolder ||
        ' with status ' || status);

      status := im_api.deletemessage(msg_id, 'INBOX');
      dbms_output.put_line('Deleted message with status ' || status);
   END LOOP;
   status := im_api.closefolder;
END;
```

Example 2: Retrieving Message Contents

The following example uses IM_API to demonstrate some of the new features in 5.1 including folder management, array access, hierarchical foldering, file logging, etc. In order to run this script, make sure the following is performed first:

1. Modify the top section so that the usernames, domain, password and log file directories are valid.
2. Populate some messages possibly with attachments in the user's mailbox.
3. Make sure the database user that the script is to be run from is granted executable on object types api_folder_c and api_template_c. If not, have them granted by DBA
4. Make sure the log file directory is listed in UTL_FILE_DIR parameter of the RDBMS.

This example can be accessed in the following directory:

```
$ORACLE_HOME/office/admin/samples/imapi_sample.sql
```

Example

```
DECLARE
  c_logdir CONSTANT VARCHAR2(80) := '/oracle/home/office/log';
  c_logname CONSTANT VARCHAR2(80) := 'api_test.log';
```

```
  c_user CONSTANT VARCHAR2(80) := 'scott';
  c_domain CONSTANT VARCHAR2(80) := 'acme.com';
  c_password CONSTANT VARCHAR2(80) := 'tiger';
  c_newpass CONSTANT VARCHAR2(80) := 'welcome';
  c_bcc CONSTANT VARCHAR2(80) := 'tom@acme.com';

  ret INTEGER;
  v_exp INTEGER;
  v_acnt INTEGER;
  v_folders api_folder_c := api_folder_c();
  v_templates api_template_c := api_template_c();
  v_msgs im_api.msg_table;
  v_uid INTEGER;
  v_hdrnames im_api.name_table;
  v_hdrvalues im_api.value_table;
  v_atts im_api.att_table;
  v_prio INTEGER;
  v_rec DATE;
  v_ex DATE;
  v_inc INTEGER;
  v_sub VARCHAR2(240);
  v_sender VARCHAR2(240);
  v_to VARCHAR2(240);
  v_cc VARCHAR2(240);
  v_from VARCHAR2(240);
  v_reply VARCHAR2(240);
  v_sent DATE;
  v_size INTEGER;
  v_newmail BOOLEAN;
  v_read INTEGER;
BEGIN
   -- this function is used to initialize logging in the
   -- $ORACLE_HOME/office/log directory.  The first parameter should
-- always be the expanded path to $ORACLE_HOME/office/log.  Also
-- passed are the log file name and log level.
   ret := im_api.enablelogging(c_logdir, c_logname, 1);

   -- Pass in username, fully qualified domain name and password to
   -- authenticate.
   ret := im_api.authenticate(c_user, c_domain, c_password);
   logpkg.log(1, 'Login as user ' || c_user || '@' || c_domain ||
     ' with status ' || ret);

   -- Change to a new password
   ret := im_api.changepassword(c_newpass);
```

```
   logpkg.log(1, 'Changed password');

   -- Create a hierarchical folder, the folder First does not have to
   -- exist.  The format should always be absolute, with / in front and
   -- no / at the end
   ret := im_api.createfolder('/First/Second');
   logpkg.log(1, 'Created folder /First/Second');

-- Rename a folder.  It only renames the deepest level.  The new
-- name can only be a string without / in it.
   ret := im_api.renamefolder('/First/Second', 'Newname');
   logpkg.log(1, 'Renamed folder to Newname');

   -- Set folder expiration to 10 days
   ret := im_api.setfolderexp('/First/Newname', 10);
   logpkg.log(1, 'Set folder expiration');

   -- Get folder expiration property
   ret := im_api.getfolderexp('/First/Newname', v_exp);
   logpkg.log(1, 'Folder /First/Newname expiration: ' || v_exp);

   -- Delete a folder, only the deepest level is deleted
   ret := im_api.deletefolder('/First/Newname');
   logpkg.log(1, 'Deleted folder /First/Newname');

   -- list all folders
   ret := im_api.listfolders(v_folders);
   logpkg.log(1, 'List of all folders:');
   FOR ii IN 1..v_folders.last LOOP
      logpkg.log(1, v_folders(ii).folder_path);
   END LOOP;

   -- list all templates, template IDs
   -- can be used to reply and forward message using
   -- ReplyWithTemplate or ForwardWithTemplate functions
   ret := im_api.listtemplates(v_templates);
   logpkg.log(1, 'List of all private template IDs and subjects:');
   FOR ii IN 1..v_templates.count LOOP
      logpkg.log(1, 'ID: ' || v_templates(ii).msg_id ||
 ' Subject: ' || v_templates(ii).subject);
   END LOOP;

   -- check for new mail
   v_newmail := im_api.hasnewmail;
   IF v_newmail THEN
```

```
   logpkg.log(1, 'You''ve got mail');
ELSE
   logpkg.log(1, 'No New Mail');
END IF;

-- Get new mail message IDs
ret := im_api.getnewmail(v_msgs);
logpkg.log(1, 'Got ' || v_msgs.count || ' new mails');
FOR ii IN 1..v_msgs.count LOOP
   logpkg.log(1, ' New mail message ID: ' || v_msgs(ii));
END LOOP;

-- Open a folder and get a list of messages.  Folder names
-- are case sensitive.
v_msgs.DELETE;
ret := im_api.openfolder('/Inbox', v_msgs);
logpkg.log(1, 'Opened folder /Inbox with ' || v_msgs.count
  || ' messages');

-- Loop through message list
FOR ii IN 1..v_msgs.count LOOP
   logpkg.log(1, '====================');
   logpkg.log(1, 'Message ID: ' || v_msgs(ii));

-- set each message to prioriy high and expiration to 3 weeks
-- later also set message as read
   ret := im_api.setmessageprops(v_msgs(ii), '/Inbox', 50,
     sysdate+21, 1);
   logpkg.log(1, 'Set message properties');

   -- get the same properties back plus received date and UID
   ret := im_api.getmessageprops(v_msgs(ii), '/Inbox', v_uid,
     v_prio, v_rec, v_ex, v_read);
   logpkg.log(1, 'UID: ' || v_uid);
   IF v_prio = 50 THEN
logpkg.log(1, 'Priority: High');
   ELSIF v_prio = 0 THEN
logpkg.log(1, 'Priority: Normal');
   ELSE
logpkg.log(1, 'Priority: Low');
   END IF;
   logpkg.log(1, 'Received: ' || v_rec);
   logpkg.log(1, 'Expire: ' || v_ex);
   IF v_read = 1 THEN
logpkg.log(1, 'Status: Read');
```

```
    ELSE
 logpkg.log(1, 'Status: Unread');
    END IF;

    -- get standard message headers
    ret := im_api.getmessagehdrs(v_msgs(ii),
v_sub, v_sender, v_to, v_cc, v_from, v_sent, v_reply, v_size);
    logpkg.log(1, 'Subject: ' || v_sub);
    logpkg.log(1, 'Sender: ' || v_sender);
    logpkg.log(1, 'TO: ' || v_to);
    logpkg.log(1, 'CC: ' || v_cc);
    logpkg.log(1, 'From: ' || v_from);
    logpkg.log(1, 'Sent: ' || v_sent);
    logpkg.log(1, 'Reply-To: ' || v_reply);
    logpkg.log(1, 'Size: ' || v_size);

    -- get messages headers that are not standard
    ret := im_api.getextendedhdrs(v_msgs(ii), v_hdrnames,
      v_hdrvalues);
    FOR jj IN 1..v_hdrnames.count LOOP
 logpkg.log(1, v_hdrnames(jj) || ' ' || v_hdrvalues(jj));
    END LOOP;

    -- get a list of unique message parts/attachments
    ret := im_api.getpartlist(v_msgs(ii), v_atts);
    FOR jj IN 1..v_atts.count LOOP
 logpkg.log(1, '-----------------');
 logpkg.log(1, 'Attachment part No: ' || v_atts(jj).part_number);
 logpkg.log(1, 'Content-Type: ' || v_atts(jj).content_type);
 IF v_atts(jj).is_binary = 'Y' THEN
    logpkg.log(1, 'Binary or Textual: Binary');
 ELSIF v_atts(jj).is_binary = 'N' THEN
    logpkg.log(1, 'Binary or Textual: Textual');
 END IF;
 logpkg.log(1, 'Attachment size: ' || v_atts(jj).att_size);
 logpkg.log(1, 'Filename: ' || v_atts(jj).att_name);
    END LOOP;
    logpkg.log(1, '-----------------');

    -- BCC the message to another user
    ret := im_api.blindcopyto(v_msgs(ii), c_bcc);
    logpkg.log(1, 'BCCed this message to ' || c_bcc);

    -- copy to another folder, note the leading /
    ret := im_api.copytofolder(v_msgs(ii), '/Inbox', '/First');
```

```
      logpkg.log(1, 'Copied this message to /First');

      -- delete this message
      ret := im_api.deletefrom(v_msgs(ii), '/Inbox');
      logpkg.log(1, 'Deleted this message');
   END LOOP;

   -- turn off logging
   ret := im_api.disablelogging;
END;
```

Using Server Side Rules

The server side rules feature in Oracle eMail Server is a mechanism for users to define automatic actions that act upon incoming messages when the messages meet certain custom-defined criteria. For example, you can set a rule to delete an e-mail when the subject contains the string "get rich fast." Actions are performed when the postman process is notifying users about new e-mail arrivals. The choices of actions and the choices of criteria to meet for a certain message are described in the following sections. Rule setup and management functionality is exposed through a set of PL/SQL functions.

Understanding Mail Rules

A rule consists of two parts, a set of conditions to be met and an action to be performed when at least one of the conditions in the set is met. The set of conditions is therefore evaluated using a short-circuit `OR` operation, meaning the evaluation stops as soon as it encounters a true condition in the set. Rules are identified by their actions, therefore if multiple actions are to be carried out when certain condition is true, then multiple rules must be defined sharing the same set of conditions.

A condition is a combination of simple clauses. A clause is a simple relational operation between a message attribute and a scalar constant value. An example of a clause can be “message size is greater than 10K”. Condition evaluation is implemented as a short-circuit `AND` operation upon all clauses; that is, the evaluation returns failure upon encountering the first failed clause.

Rule Management API

Oracle eMail Server provides two sets of APIs to manage rules. One set of APIs is intended to be used by users or application integrators that act as users. This set of

APIs are embedded in the IM_API package, taking advantage of facilities provided in the package such as authentication, logging, and access to folders and templates. The other set of APIs is intended to be used by administrators that have the privilege of directly accessing the e-mail database and monitoring users. The former is ideal to build applications that enables users to setup e-mail filters for themselves, or integrate with other applications. The latter is best suited for administrative scripting and setting up system-wide rules for the entire server.

User Rules Management API Function Reference Object Types

User rules management API makes uses of Oracle object types to retrieve and store rules information. The following object types are described in this section:

- api_rule_t
- api_rule_c
- api_condition_t
- api_condition_c

api_rule_t

The declaration for this type is:

```
CREATE OR replace TYPE api_rule_t AS object (
   rule_id       INTEGER,
   rule_name     VARCHAR2(80),
   active        VARCHAR2(1),
   action_id     INTEGER,
   num_param     INTEGER,
   vchar_param   VARCHAR2(240),
   is_and        INTEGER);
```

Description

The object type represents a rule object.

Member Variables

Rule_id	A number uniquely identifies a rule in the system
Rule_name	A descriptive, meaningful name for a rule

Active	A value 'A' means the rule is enabled, a value 'D' means the rule is disabled
Action_ID	The numerical ID corresponding to the type of actions. The values are: 0: No action (placeholder) ACTION_ALT_RECIP: Forward to an alternate recipient (recipient = vchar_param) ACTION_DELETE: Delete message ACTION_MOVE: Move to folder (folder_id = num_param) ACTION_COPY: Copy to folder (folder_id = num_param) ACTION_REPLY: Generate reply with template (template_id = num_param) ACTION_RUN_PROC: Invoke a PL/SQL procedure (procedure name = vchar_param) ACTION_EXPIRE: Set expiration on message (new expiration date = vchar_param) ACTION_SET_PRIO: Alter priority (new priority = num_param) ACTION_FORWARD: Forward to a recipient with template (template_id = num_param) ACTION_STOP: Stop further rule execution for this message ACTION_BOUNCE: Bounce the message back to the sender
Num_param:	The numerical parameter required in some of the actions above
Vchar_param:	The VARCHAR2 parameter required in some of the actions above
Is_and:	A value 1 means all the conditions should be ANDed together, a value of 0 means all theconditions should be ORed together

api_rule_c

The declaration for this type is:

```
CREATE OR replace TYPE api_rule_c AS TABLE OF api_rule_t;
```

Description

This type is a nested table of API_RULE_T, representing a list of rules.

api_condition_t

The declaration for this type is:

```
CREATE OR replace TYPE api_condition_t AS object (
   rule_id       INTEGER,
```

```
attr_id       INTEGER,
op            INTEGER,
vchar_value   VARCHAR2(255));
```

Description

This object type represents a condition object. Every condition object has to be associated with a rule using its rule_id member variable.

Member Variables

Rule_id	ID of the rule which owns this condition object
Attr_id	Message attribute type that includes the following values: ATTR_PRIORITY: Priority ATTR_SNDR_NAME; Sender Name ATTR_MSG_TYPE: Message Type ATTR_MSG_SIZE: Message Size ATTR_ATT_FLAG: Attachment Flag ATTR_EXP_DATE: Expiration Flag ATTR_BCC_RECIP: BCC recipient Flag ATTR_SUBJECT: Message Subject ATTR_TO_RECIP: To Recipient ATTR_CC_RECIP: CC Recipient ATTR_FROM_STR: From header ATTR_SENT_DATE: Sent Date ATTR_MSG_ID: internal message ID ATTR_RULE_STATUS: the status code of the last rule execution ATTR_OVER_QUOTA: the over_quota status of the current messages

Op	Operator code that includes the following values: COMP_EQ: = (equal) COMP_GT: > (greater than) COMP_GE: >= (greater than or equal) COMP_LT: < (less than) COMP_LE: <= (less than or equal) COMP_IS_NULL: IS NULL COMP_IS_NOT_NULL: IS NOT NULL COMP_NE: <> (not equals) COMP_STARTS_WITH: LIKE 'X%' (starts with) COMP_ENDS_WITH: LIKE '%X' (ends with) COMP_CONTAINS: LIKE '%X%' (contains - case insensitive) COMP_NOT_CONTAIN: NOT LIKE '%X%' (not contains) COMP_IS_TRUE: test for TRUE COMP_IS_FALSE: test for FALSE
Vchar_value	A constant value that the message attribute is compared against. This constant value should be cast into VARCHAR2 if it is of other data types.

api_condition_c

The declaration for this type is:

```
CREATE OR replace TYPE api_condition_c IS TABLE OF api_condition_t;
```

Description

This type is a nested table of API_CONDITION_T, representing a list of conditions.

Functions

The following functions are described in this section:

- ListRules Function
- EnableRule Function
- DisableRule Function
- DeleteRule Function
- OrderRules Function
- SaveRuleDetail Function

ListRules Function

This function retrieves all the rule objects along with condition objects for the current user. You need to call IM_API.AUTHENTICATE before using these functions.

Syntax for the ListRules Function

```
FUNCTION ListRules(
   p_rules  IN OUT  api_rule_c,
   p_conds  IN OUT  api_condition_c)
RETURN INTEGER;
```

Output of the ListRules Function

Table 2–83

Output	Description
p_rules	List of rule objects
p_conds	List of conditions objects that the rule object refers to

Returns of the ListRules Function

Table 2–84

Returned Message	Description
IMAPI_OK	Success
IMAPI_ERR_FAUTH	Not authenticated. This means the AUTHENTICATE function has not been called.

EnableRule Function

This function marks a given rule as enabled.

Syntax for the EnableRule Function

```
FUNCTION EnableRule(
   p_ruleid  IN  INTEGER)
RETURN INTEGER;
```

Input for the EnableRule Function

Table 2–85

Input	Description
p_ruleid	Rule ID of the rule to be enabled

Returns of the EnableRule Function

Table 2–86

Returned Message	Description
IMAPI_OK	Success
IMAPI_ERR_FAUTH	Not authenticated. This means the AUTHENTICATE function has not been called.

DisableRule Function

This function marks a given rule as disabled. A disabled rule still exists in the database but will not be executed.

Syntax for the DisableRule Function

```
FUNCTION DisableRule(
   p_ruleid  IN  INTEGER)
RETURN INTEGER;
```

Input for the DisableRule Function

Table 2–87

Input	Description
p_ruleid	Rule ID of the rule to be disabled

Returns of the DisableRule Function

Table 2–88

Returned Message	Description
IMAPI_OK	Success

Table 2–88

Returned Message	Description
IMAPI_ERR_FAUTH	Not authenticated. This means the AUTHENTICATE function has not been called.

DeleteRule Function

This function deletes a given rule from the system.

Syntax for the DeleteRule Function

```
FUNCTION DeleteRule(
   p_ruleid  IN  INTEGER)
RETURN INTEGER;
```

Input for the DeleteRule Function

Table 2–89

Input	Description
p_ruleid	Rule ID of the rule to be deleted

Returns of the DeleteRule Function

Table 2–90

Returned Message	Description
IMAPI_OK	Success
IMAPI_ERR_FAUTH	Not authenticated. This means the AUTHENTICATE function has not been called.

OrderRules Function

This function takes an ordered collection of rule objects in an array and reorders the rule execution orders in the system accordingly.

Syntax for the OrderRules Function

```
FUNCTION ListRules(
   p_rules  IN  api_rule_c)
RETURN INTEGER;
```

Input for the OrderRules Function

Table 2–91

Input	Description
p_ruleid	Ordered list of rule objects

Returns of the OrderRules Function

Table 2–92

Returned Message	Description
IMAPI_OK	Success
IMAPI_ERR_FAUTH	Not authenticated. This means the AUTHENTICATE function has not been called.

SaveRuleDetail Function

This function takes a rule object and its associated condition objects and saves it in the system. If the rule object already exists, then it modifies the existing rule according to the new input. If it does not exist, then it creates a new rule based on input.

Syntax for the SaveRuleDetail Function

```
FUNCTION SaveRuleDetail(
   p_rule   IN OUT  api_rule_t,
   p_conds  IN      api_condition_c)
RETURN INTEGER;
```

Input for the SaveRuleDetail Function

Table 2–93

Input	Description
p_rule	The rule object to be saved. If this is a modification to an existing rule, then the rule_id member of the object should contain the original rule ID. If this is a new rule to be created, then the rule_id member should be set to 0.
p_conds	The condition object list that contains all the conditions needed for the rule to be saved.

Output of the SaveRuleDetail Function

Table 2–94

Output	Description
p_rule	If this is a new rule, then the modified rule object will be returned containing the new rule ID. This rule ID can be use later to perform more modifications.

Returns of the SaveRuleDetail Function

Table 2–95

Returned Message	Description
IMAPI_OK	Success
IMAPI_ERR_FAUTH	Not authenticated. This means the AUTHENTICATE function has not been called.

Administrator Rules Management API Function Reference

The following functions are described in this section:

- start_condition Function
- add_clause Function
- create_rule Function
- add_cond Function
- delete_rule Function
- toggle_rule Function

start_condition Function

The start_condition function starts a new condition by returning a unique condition ID for you to make subsequent modifications to it.

Syntax for the start_condition Function

```
FUNCTION start_condition RETURN INTEGER;
```

Returns of the start_condition Function

Table 2–96

Return	Description
condid	Condition ID of the condition to be used in subsequent calls

add_clause Function

The add_clause function adds a clause into an existing condition identified by the condition ID.

Syntax for the add_clause Function

```
procedure add_clause(
   condid   IN      INTEGER,
   attrid   IN      INTEGER,
   compid   IN      INTEGER,
   valuen   IN      INTEGER,
   valuev   IN      VARCHAR2,
   valued   IN      DATE,
   orderno  IN      INTEGER,
   status   IN OUT  INTEGER);
```

Input for the add_clause Function

Table 2–97

Input	Description
condid	Existing condition ID, typically derived from a call to the start_condition function.

Table 2–97

Input	Description
attrid	Attribute ID that identifies a message attribute, used in the relational comparison. Attribute IDs are defined as follows: ATTR_PRIORITY (priority) ATTR_SNDR_NAME (sender name) ATTR_MSG_TYPE (message type) ATTR_MSG_SIZE (message size) ATTR_ATT_FLAG (attachment flag) ATTR_EXP_DATE (expiration flag) ATTR_BCC_RECIP (BCC recipient flag) ATTR_SUBJECT (message subject) ATTR_TO_RECIP (To recipient) ATTR_CC_RECIP (CC recipient) ATTR_FROM_STR (From header) ATTR_SENT_DATE (sent date) ATTR_MSG_ID (message ID) ATTR_RULE_STATUS (last rule execution status code) ATTR_OVER_QUOTA: over-quota status
compid	Comparison ID that identifies the type of relational operation to be performed in this clause. Comparison IDs are defined as follows: COMP_EQ (= (equals)) COMP_GT (> (greater than)) COMP_GE (>= (greater than or equal)) COMP_LT (< (less than)) COMP_LE (<= (less than or equal)) COMP_IS_NULL (IS NULL) COMP_IS_NOT_NULL (IS NOT NULL) COMP_NE (<> (not equals)) COMP_STARTS_WITH (LIKE 'X%' (starts with)) COMP_ENDS_WITH (LIKE '%X' (ends with)) COMP_CONTAINS (LIKE '%X%' (contains - case (insensitive)) COMP_NOT_CONTAIN (NOT LIKE '%X%' (not (contains)) COMP_IS_TRUE: test for TRUE COMP_IS_FALSE: test for FALSE
valuen	Holds a numerical value to be compared against numerical attributes such as priority or message size.
valuev	Holds a character string value to be compared against string based attributes such as send name or subject.

Table 2–97

Input	Description
valued	Holds an Oracle date value to be compared against a date attribute like expiration date, etc.
orderno	Order numbers should be maintained by the caller to ensure the order of clause evaluation when the condition is to be checked.

Output for the add_clause Function

Table 2–98

Output	Description
status	Stores the return status of this procedure. A non-zero value indicates an error.

create_rule Function

The create_rule function creates a rule based on a starting condition passed in by the caller.

Syntax for the create_rule Function

```
FUNCTION create_rule(
   username  IN      VARCHAR2,
   domain    IN      VARCHAR2,
   name      IN      VARCHAR2,
   condid    IN      INTEGER,
   actnid    IN      INTEGER,
   infon1    IN      INTEGER,
   infon2    IN      INTEGER,
   infov     IN      VARCHAR2,
   status    IN OUT  INTEGER)
RETURN INTEGER;
```

Input for the create_rule Function

Table 2–99

Input	Description
username	Identifies the user for which the rule should be created

Table 2–99

Input	Description
domain	Fully qualified domain of the user
name	Character string representing this rule (for convenience)
condid	Existing condition ID, serving as the first condition for this rule
actnid	Action ID that identifies which action to perform upon a positive evaluation of one of the conditions in the rule. Action IDs are defined as follows: 0 (no action (placeholder)) ACTION_ALT_RECIP (alternate recipient form of (notification (recipient = (infon1))) ACTION_DELETE (delete message) ACTION_MOVE (move to folder (folder_id = (infon1)) ACTION_COPY (copy to folder (folder_id = (infon1)) ACTION_REPLY (generate reply with (template (template_id = (infon1)) ACTION_RUN_PROC (Invoke a PL/SQL procedure (procedure name = infov, parameter chain looks like: msg_id, username, domain, infon1, infon2, rstatus) ACTION_EXPIRE (set expiration on message (new expiration = infov)) ACTION_SET_PRIO (alter priority (new priority = infon1)) ACTION_FORWARD (forward to a recipient with template (template_id = num_param)) ACTION_STOP (stop further rule execution for this message) ACTION_BOUNCE (Bounce the message back to the sender)
infon1	Holds a numerical value that is only meaningful to some specific actions. For example, ACTION_MOVE (move to folder) requires that infon1 be used to pass in the destination folder ID.
infon2	Holds a second numerical value that is only meaningful if the specific action requires it. For example, ACTION_RUN_PROC (invoking a PL/SQL procedure) can require a meaningful value in infon2, depending on the procedure to be called.

Table 2–99

Input	Description
infov	Holds a character string value that is only meaningful to specific actions. For example, ACTION_ALT_RECIP (submit to alternative recipient) requires that infov contain the recipient address in order to forward the message.

Output of the create_rule Function

Table 2–100

Output	Description
status	The return status of this procedure. A non-zero value indicates an error.

Returns of the create_rule Function

A unique rule ID that identifies the rule just created.

add_cond Function

Syntax for the add_cond Function

```
procedure add_cond(
   ruleid  IN      INTEGER,
   ondid   IN      INTEGER,
   tatus   IN OUT  INTEGER);
```

Description

The add_cond function adds a condition into an existing rule identified by the rule ID.

Input

ruleid	An existing rule ID, should be originally coming from a call to create_rule.
condid	An existing condition ID to be inserted in the rule.

Output

status	The return status of this procedure. A non-zero value indicates an error.

delete_rule Function

Syntax for delete_rule Function

```
procedure delete_rule(
   ruleid  IN      INTEGER,
   status  IN OUT  INTEGER);
```

Description

The delete_rule function deletes a rule from the system on behalf of a user. If the condition IDs referenced by the rule no longer used in the system, delete the condition ID too. If deleting this rule results in this user having no rules at all, disable the user preference that specifies the user has rules.

Input

ruleid	An existing rule ID to be deleted

Output

status	Return status of this procedure. A non-zero value indicates an error.

toggle_rule Function

Syntax for toggle_rule Function

```
procedure toggle_rule(
   ruleid  IN      INTEGER,
   active  IN      BOOLEAN,
   status  IN OUT  INTEGER);
```

Description

The toggle_rule function toggles the active flag on this rule. A rule can be disabled or enabled by this call. Disabled rules stay on the system until the delete_rule function is called.

Input

ruleid	Existing rule ID to be disabled / enabled.
active	A Boolean value. When set to true, the rule is enabled, otherwise the rule is disabled.

Output

status	The return status of this procedure. A non-zero value indicates an error.

Viewing a List of Rules

The view OM_USER_RULES can be queried to obtain a list of all the rules defined for users. For example, after running the sample rules script in the following section to create rules, the following SQL commands provide a list of the rules created:

```
SQL> select * from om_user_rules where name='SCOTT';
NAME                                RULE_ID Rule_State
------------------------------ ---------- ----------
COMPARISON
--------------------------------------------------------------------
Rule_Action
--------------------------------------------------------------------
SCOTT                                       1010 Active
Subject contains (LIKE '%X%') 'get rich fast'
Delete message

SCOTT                                       1010 Active
Message size greater than (>) '10000'
Delete message

SCOTT                                       1010 Active
Subject contains (LIKE '%X%') 'test message'
Delete message
```

Writing Customized Procedure for ACTION_RUN_PROC

If a rule is defined to call an external PL/SQL procedure as its action, the procedure must be written with the following restrictions:

1. The procedure name must be a valid name referenceable by the mail server. Typically the procedure must be granted executable by public, or be granted to user oo specifically. The name must be prefixed by account name that owns the procedure, and may optionally be appended by a database link name if the procedure resides on a remote database.
2. The procedure must take the following parameters in order:

```
msg_id   IN integer,
username IN varchar2,
domain   IN varchar2,
ninfo1   IN integer,
ninfo2   IN integer,
rstatus  OUT integer
```

The msg_id, username, and domain supply the basic information about the current message and its current recipient. The ninfo1 and ninfo2 is used to hold additional customized input to make a procedure reusable in different rules. The rstatus needs to be returned by the procedure and can be used subsequently in rules based on the attribute ATTR_RULE_STATUS.

3. The procedure cannot perform commit or rollback operations.

Server Side Rules Sample Script

Example 1: Using User Rule Management API

The following example uses IM_API to setup two rules, and also to demonstrate other functionalities including rule retrieval, re-ordering and deletion. Before running the script, make sure the following is performed first:

1. Modify the constant declaration section so that the usernames, domain, password, folder names and log file directories are valid.
2. Make sure the database user that the script is to be run from is granted executable on object types api_rule_c, api_rule_t and api_condition_c. If not, have them granted by DBA.

3. Make sure the log file directory is listed in UTL_FILE_DIR parameter of the RDBMS.

This example can be accessed in the following directory:

```
$ORACLE_HOME/office/admin/samples/rules_sample.sql

DECLARE
   c_user CONSTANT VARCHAR2(80) := 'scott';
   c_domain CONSTANT VARCHAR2(80) := 'acme.com';
   c_password CONSTANT VARCHAR2(80) := 'tiger';
   c_logdir CONSTANT VARCHAR2(80) := '/oracle/home/office/log';
   c_logname CONSTANT VARCHAR2(80) := 'rule_test.log';
   c_folder CONSTANT VARCHAR2(80) := '/Mailing lists';

   v_rules api_rule_c := api_rule_c();
   v_folders api_folder_c := api_folder_c();
   v_conds api_condition_c := api_condition_c();
   v_res INTEGER;
   v_yes VARCHAR2(10);
   v_andor VARCHAR2(10);
   v_rule api_rule_t;
   v_jj INTEGER := 1;
   v_id INTEGER;
BEGIN
   -- setup logging and login as a mail user
   v_res := im_api.enablelogging(c_logdir, c_logname, 1);
   v_res := im_api.authenticate(c_user, c_domain, c_password);

   -- create the first rule:
   -- if Subject: header contains 'for sale' and size > 10000
   --   delete;
   v_conds.extend(2);
   v_conds(1) := api_condition_t(0, mail_rules.attr_subject,
     mail_rules.comp_contains, 'for sale');
   v_conds(2) := api_condition_t(0, mail_rules.attr_msg_size,
     mail_rules.comp_gt, '10000');

   v_rule := api_rule_t(0, 'Delete Junk Mail', 'A',
     mail_rules.action_delete, NULL, NULL, 1);

   v_res := im_api.saveruledetail(v_rule, v_conds);
   logpkg.log(1, 'Created rule ID ' || v_rule.rule_id);

   -- create another rule:
   -- if TO: header does not contain myself
```

```
 --   move to folder '/Mailing lists';
 v_conds.DELETE(2);
 v_conds(1) := api_condition_t(0, mail_rules.attr_to_recip,
   mail_rules.comp_not_contain, c_user);

 -- find out folder ID for '/Mailing lists'
 v_res := im_api.listfolders(v_folders);
 FOR ii IN 1..v_folders.last LOOP
    IF v_folders(ii).folder_path = c_folder THEN
v_id := v_folders(ii).folder_id;
EXIT;
    END IF;
 END LOOP;

 v_rule := api_rule_t(0, 'Save list mails', 'A',
   mail_rules.action_move, v_id, NULL, 1);

 v_res := im_api.saveruledetail(v_rule, v_conds);
 logpkg.log(1, 'Created rule ID ' || v_rule.rule_id);

 v_rules.DELETE;
 -- now retrieve the rules
 v_res := im_api.listrules(v_rules, v_conds);
 logpkg.log(1, 'Retrieved ' || v_rules.count || ' rules');

 FOR ii IN 1..v_rules.count LOOP
    logpkg.log(1, '-------------------------------------');
    logpkg.log(1, 'Rule name: ' || v_rules(ii).rule_name);
    logpkg.log(1, 'Rule ID: ' || v_rules(ii).rule_id);
    IF v_rules(ii).active = 'A' THEN
v_yes := 'yes';
    ELSE
v_yes := 'no';
    END IF;
    logpkg.log(1, 'Active: ' || v_yes);

    -- list conditions
    IF v_rules(ii).is_and = 1 THEN
v_andor := '(AND)';
    ELSE
v_andor := '(OR)';
    END IF;
    logpkg.log(1, 'Condition' || v_andor || ':');
    WHILE v_jj <= v_conds.count
AND v_conds(v_jj).rule_id = v_rules(ii).rule_id LOOP
```

```
  logpkg.log(1,
    im_api.rule_attrtab(v_conds(v_jj).attr_id) || ' ' ||
    im_api.rule_optab(v_conds(v_jj).op) || ' ' ||
    v_conds(v_jj).vchar_value);
  v_jj := v_jj + 1;
      END LOOP;

      -- print action
      logpkg.log(1, 'Action: ');
      IF v_rules(ii).num_param IS NOT NULL THEN
  logpkg.log(1, im_api.rule_actiontab(v_rules(ii).action_id) ||
    ' ' || v_rules(ii).num_param);
      ELSIF v_rules(ii).vchar_param IS NOT NULL THEN
  logpkg.log(1, im_api.rule_actiontab(v_rules(ii).action_id) ||
    ' ' || v_rules(ii).vchar_param);
      ELSE
  logpkg.log(1, im_api.rule_actiontab(v_rules(ii).action_id));
      END IF;
   END LOOP;
   logpkg.log(1, '----------------------------------');

   -- reorder the rules, make the first rule become the last
   v_rules.extend;
   v_rules(v_rules.count) := v_rules(1);
   FOR ii IN 1..v_rules.count - 1 LOOP
      v_rules(ii) := v_rules(ii+1);
   END LOOP;
   v_rules.DELETE(v_rules.count);
   v_res := im_api.orderrules(v_rules);
   logpkg.log(1, 'Re-ordered rules');

   -- Delete the first rule
   v_res := im_api.deleterule(v_rules(1).rule_id);
   logpkg.log(1, 'Deleted a rule');

   v_res := im_api.disablelogging;
END;
```

Example 2: Using Admin Rule Management API

The following example uses MAIL_RULES package to setup a number of rules, demonstrating the variety of ways to create rule actions and conditions. Before running the script, make sure the following is performed first:

1. Modify the constant declaration section so that the usernames, domain, password, folder names and log file directories are valid.
2. Make sure the database user that the script is to be run from is granted executable on object types api_rule_c, api_rule_t and api_condition_c. If not, have them granted by DBA
3. Make sure the log file directory is listed in UTL_FILE_DIR parameter of the RDBMS.

This example can be accessed in the following directory:

```
$ORACLE_HOME/office/admin/samples/rules_sample.sql
```

```
-- simple procedure used by custom PL/SQL rule callout
CREATE OR replace PROCEDURE process_order(msg_id   IN INTEGER,
       username  IN VARCHAR2,
       domain  IN VARCHAR2,
       ninfo1   IN INTEGER,
       ninfo2   IN INTEGER,
       rstatus  OUT INTEGER)   IS
   c_logdir CONSTANT VARCHAR2(80) := '/oracle/home/office/log';
   c_logfile CONSTANT VARCHAR2(80) := 'rule_test.log';
BEGIN
   logpkg.init(c_logdir, c_logfile, 5);
   logpkg.log(1, 'Processed message detail:');
   logpkg.log(1, '-------------------------');
   logpkg.log(1, 'Message ID: ' || msg_id);
   logpkg.log(1, 'User name: ' || username);
   logpkg.log(1, 'Domain name: ' || domain);
   logpkg.log(1, 'Parameter 1: ' || ninfo1);
   logpkg.log(1, 'Parameter 2: ' || ninfo2);
   logpkg.log(1, '-------------------------');
   logpkg.cleanup;
   rstatus := 0;
END;
/

-- Setting up server side rules using MAIL_RULES package
DECLARE
   c_user CONSTANT VARCHAR2(30) := 'scott';
   c_domain CONSTANT VARCHAR2(80) := 'acme.com';
   c_folder1 CONSTANT VARCHAR2(80) := 'Important Events';
   c_folder2 CONSTANT VARCHAR2(80) := 'Announcement Records';
   c_recip CONSTANT VARCHAR2(80) := 'tom@acme.com';
   c_template CONSTANT VARCHAR2(80) := 'ORDER RECEIVED';
```

```
   ruleid     INTEGER;
   condid     INTEGER;
   status     INTEGER;
   folderid   INTEGER;
   templateid INTEGER;
BEGIN
   -- create rule 1:
   -- Delete message if subject contains 'for sale'
   -- and size > 10000 bytes,
   -- or if subject contains 'get paid to surf'.
   condid:= mail_rules.start_condition;

   -- subject contains 'for sale'
   mail_rules.add_clause(condid, MAIL_RULES.ATTR_SUBJECT,
     MAIL_RULES.COMP_CONTAINS, 0, 'for sale', NULL, 1, status);

   -- and message size > 10000 bytes; or ..
   mail_rules.add_clause(condid, MAIL_RULES.ATTR_MSG_SIZE,
     MAIL_RULES.COMP_GT, 10000, NULL, NULL, 2, status );

   -- create a rule
   ruleid := mail_rules.create_rule(c_user, c_domain,
     'Delete Junk Mail', condid, MAIL_RULES.ACTION_DELETE, 0, 0,
     NULL, status);

   -- create a new condition, needs a new condid
   condid:= mail_rules.start_condition;

   -- .. or, subject contains 'get paid to surf'
   mail_rules.add_clause(condid, MAIL_RULES.ATTR_SUBJECT,
     MAIL_RULES.COMP_CONTAINS, 0, 'get paid to surf', NULL, 1, status);

   -- add an 'OR'ed condition to the existing rule
   mail_rules.add_cond(ruleid, condid, status);

   -- Create rule 2: Send message to alternative recipient if subject
   -- contains 'Meeting anouncement'.
   condid:= mail_rules.start_condition;

   -- subject contains 'Meeting anouncement'
   mail_rules.add_clause(condid, MAIL_RULES.ATTR_SUBJECT,
     MAIL_RULES.COMP_CONTAINS, 0, 'Meeting announcement', NULL,
     1, status);
```

```
    ruleid := mail_rules.create_rule(c_user, c_domain,
      'BCC announcements', condid, MAIL_RULES.ACTION_ALT_RECIP, 0,
      0, c_recip, status);

    -- Create rule 3: Move message to folder 'Important Events'
    -- if message priority is high (the number 50)
    condid:= mail_rules.start_condition;

    -- Message priority = 50
    mail_rules.add_clause(condid, MAIL_RULES.ATTR_PRIORITY,
      MAIL_RULES.COMP_EQ, 50, NULL, NULL, 1, status);

    -- this rule requires folder ID that we can either use IM_API
    -- or directly query the mail database.  Since this is a privileged
    -- script, we'll directly query the database
    select FOLDER_ID into folderid from OM_folder
      where disp_name = c_folder1 AND creator =
      (SELECT objectid FROM ds_account WHERE name=upper(c_user));

    ruleid := mail_rules.create_rule(c_user, c_domain,
      'Prioritize mail', condid, MAIL_RULES.ACTION_MOVE, folderid,
      0, NULL, status);

-- Create rule 4: Copy message to folder 'Announcement Records'
-- if message subject contains 'Announcement'
    condid:= mail_rules.start_condition;

    -- subject contains 'Announcement'
    mail_rules.add_clause(condid, MAIL_RULES.ATTR_SUBJECT,
      MAIL_RULES.COMP_CONTAINS, 0, 'Announcement', NULL, 1, status);

    select FOLDER_ID into folderid from OM_folder
      where disp_name = c_folder2 AND creator =
      (SELECT objectid FROM ds_account WHERE name=upper(c_user));

    ruleid := mail_rules.create_rule(c_user, c_domain, 'Keep a record',
      condid, MAIL_RULES.ACTION_COPY, folderid, 0, NULL, status);

-- Create rule 5: Generate reply with template 'ORDER RECEIVED' if
-- the subject of the original message contains 'ORDER ENTERED: '
    condid:= mail_rules.start_condition;

    -- Subject contains 'ORDER ENTERED: '
    mail_rules.add_clause(condid, MAIL_RULES.ATTR_SUBJECT,
      MAIL_RULES.COMP_CONTAINS, 0, 'ORDER ENTERED:', NULL, 1, status);
```

```
  -- get the template ID from the database.  This example uses
  -- direct query to the database for simplicity
  select MSG_ID into templateid from OM_template
    where TEMPL_NAME = c_template AND templ_owner =
    (SELECT objectid FROM ds_account WHERE name=upper(c_user));

  ruleid := mail_rules.create_rule(c_user, c_domain, 'Auto-reply',
    condid, MAIL_RULES.ACTION_REPLY, templateid, 0, NULL, status);

  -- Create rule: Invoke PL/SQL procedure 'PROCESS_ORDER' if the
  -- subject of the original message contains 'ORDER : '
  condid:= mail_rules.start_condition;

  -- subject contains 'ORDER ENTERED: '
  mail_rules.add_clause(condid, MAIL_RULES.ATTR_SUBJECT,
    MAIL_RULES.COMP_CONTAINS, 0, 'ORDER ENTERED:', NULL, 1, status);

  ruleid := mail_rules.create_rule(c_user, c_domain, 'Process order',
    condid, MAIL_RULES.ACTION_RUN_PROC, 0, 0, 'PROCESS_ORDER',
    status);

  -- Create rule: Alter priority of message to high (the number 50)
  -- if subject contains 'Meeting'
  -- create a new condition, needs a new condid
  condid:= mail_rules.start_condition;

  -- Subject contains 'Meeting'
  mail_rules.add_clause(condid, MAIL_RULES.ATTR_SUBJECT,
    MAIL_RULES.COMP_CONTAINS, 0, 'Meeting', NULL, 1, status);

  ruleid := mail_rules.create_rule(c_user, c_domain, 'Mark meetings',
    condid, MAIL_RULES.ACTION_SET_PRIO, 50, 0, NULL, status);

  -- By default rules are created enabled.  Disable the last rule
  mail_rules.toggle_rule(ruleid, FALSE, status);

  -- Enable the last rule
  mail_rules.toggle_rule(ruleid, TRUE, status);

  -- To delete last rule
  mail_rules.delete_rule(ruleid, status);
END;
```

Using the DAPLS Package

A PL/SQL package named DAPLS is installed as part of Oracle eMail Server to enable PL/SQL programmers to create user accounts and perform other administrative tasks. This package supports some but not all IOFCMGR commands.

	Supported	Not Supported
Object Management	insert, update, delete	fetch
Security	grant, revoke, setpwd user	check, setpwd admin, setpwd database
Process Management	startup, shutdown, refresh, register, deregister, modify, cleanup	show
Queue Management	bounce, reroute, resume, suspend	display
Node Management		boot, move, subscribe, unsubscribe
Utility	import, export, execute, verify, replicate, publish, implst, explst, reset	
General		help, connect, host, exit, echo, rem, list, change, run, spool, commit, rollback, pause, whoami, autocommit

Prerequisites and Setup

The DAPLS package is automatically loaded with the installation of Oracle eMail Server. This package works together with the external procedure call feature of Net8. To use the DAPLS package you must have your Net8 listener running and configured to listen for external procedure calls.

> **See Also:** *Oracle Net8 Administrator's Guide* for more information on Configuring Net8 for External Procedures

DAPLS Procedure Reference

This packages defines the following PL/SQL exceptions:

```
invalid_admin_credential EXCEPTION;
invalid_messaging_domain EXCEPTION;
invalid_command          EXCEPTION;
```

ACCT_CREATE Procedure

This procedure is deprecated. The functionality of this procedure is available from the ADMIN_EXEC procedure.

ACCT_UPDATE Procedure

This procedure is deprecated. The functionality of this procedure is available from the ADMIN_EXEC procedure.

ACCT_DELETE Procedure

This procedure is deprecated. The functionality of this procedure is available from the ADMIN_EXEC procedure.

PASS_UPDATE Procedure

This procedure is deprecated. The functionality of this procedure is available from the ADMIN_SETPWD procedure.

SET_ADMIN_ID Procedure

The SET_ADMIN_ID procedure sets the administrator ID that will be used for authentication by subsequent calls to the ADMIN_EXEC and ADMIN_SETPWD procedures. This ID is typically "admin" but may also be the account ID of any messaging account that has administrative privileges for the target administrative messaging domain.

Syntax for the SET_ADMIN_ID Procedure

```
PROCEDURE SET_ADMIN_ID (admin_id_in IN VARCHAR2)
```

Input for the SET_ADMIN_ID Procedure

Input	Description
admin_id_in	contains the administrator ID that will be used for authentication by subsequent calls to the ADMIN_ EXEC and ADMIN_SETPWD procedures

SET_ADMIN_PASSWORD Procedure

The SET_ADMIN_PASSWORD procedure sets the administrator password that will be used for authentication by subsequent calls to the ADMIN_EXEC and ADMIN_ SETPWD procedures. This password must be the correct password for the administrator account specified by calling SET_ADMIN_ID.

Syntax for the SET_ADMIN_PASSWORD Procedure

```
PROCEDURE SET_ADMIN_PASSWORD(admin_password_in IN VARCHAR2)
```

Input for the SET_ADMIN_PASSWORD Procedure

Input	Description
admin_password_in	contains the administrator password that will be used for authentication by subsequent calls to the ADMIN_EXEC and ADMIN_SETPWD procedures

SET_SERVICE_ADDRESS Procedure

The SET_SERVICE_ADDRESS procedure sets the TNS service address used for establishing TNS connections by subsequent calls to the ADMIN_EXEC and ADMIN_SETPWD procedures. This service address should refer to the eMail Server database and it should be possible for the TNS listener to resolve the address by examining the tnsnames.ora file or by using any other appropriate service name resolution facility, e.g. Oracle Names.

Syntax for the SET_SERVICE_ADDRESS Procedure

```
PROCEDURE SET_SERVICE_ADDRESS(service_address_in IN VARCHAR2)
```

Input for the SET_SERVICE_ADDRESS Procedure

Input	Description
service_address_in	The service address of the eMail Server database

SET_MESSAGING_DOMAIN Procedure

The SET_MESSAGING_DOMAIN procedure sets the eMail Server domain used by subsequent calls to the ADMIN_EXEC and ADMIN_SETPWD procedures. The messaging domain typically has the form "<company-name>.com"

Syntax for the SET_MESSAGING_DOMAIN Procedure

```
PROCEDURE SET_MESSAGING_DOMAIN(messaging_domain_in IN VARCHAR2)
```

Input for the SET_MESSAGING_DOMAIN Procedure

Input	Description
messaging_domain_id_in	The eMail Server domain used by subsequent calls to the ADMIN_EXEC and ADMIN_SETPWD procedures

ADMIN_EXEC Procedure

The ADMIN_EXEC procedure allows most IOFCMGR command lines to be performed. Before ADMIN_EXEC is called, the procedures SET_ADMIN_ID, SET_ADMIN_PASSWORD, SET_SERVICE_ADDRESS, and SET_MESSAGING_DOMAIN must be called to initialized the administrative credentials and service address information that will be used when the command line is performed. After this procedure is called, the status_inout parameter will contain a status code indicating whether the command line was performed successfully or not.

The status_inout parameter MUST be initialized to 0 before admin_exec() is called.

An example command line is "insert person username=chad uanode=esnode"

Not all OOMGR command lines are supported by the admin_exec() procedure.

Syntax for the ADMIN_EXEC Procedure

```
PROCEDURE ADMIN_EXEC(
command      IN     VARCHAR2,
```

```
status_inout IN OUT BINARY_INTEGER)
```

Input for the ADMIN_EXEC Procedure

Input	Description
command	OOMGR command line

Output of the ADMIN_EXEC Procedure

Output	Description
status_inout	command return code; 0 for success and not 0 for failure

Exceptions thrown by the ADMIN_EXEC Procedure

Exception	Description
invalid_admin_credential	thrown if the admin_id or admin_password is equal to NULL
invalid_messaging_domain	thrown if the messaging_domain is equal to NULL
invalid_command	thrown if the command is equal to NULL

ADMIN_SETPWD Procedure

The ADMIN_SETPWD procedure allows the password for an eMail Server user account to be changed. Before ADMIN_SETPWD is called, the procedures SET_ADMIN_ID, SET_ADMIN_PASSWORD, SET_SERVICE_ADDRESS, and SET_MESSAGING_DOMAIN must be called to initialized the administrative credentials and service address information that will be used when the user account password is set. After this procedure is called, the status_inout parameter will contain a status code indicates whether the password was changed successfully or not.

The status_inout parameter MUST be initialized to 0 before admin_exec() is called.

The service address must resolve to the user's home node database. This is only of concern in a multi-server installation.

Syntax for the ADMIN_SETPWD Procedure

```
PROCEDURE ADMIN_SETPWD(
user_id      IN     VARCHAR2,
user_pass    IN     VARCHAR2,
status_inout IN OUT BINARY_INTEGER)
```

Input for the ADMIN_SETPWD Procedure

Input	Description
user_id	ID of an eMail Server user account
user_pass	new password for the user

Output of the ADMIN_SETPWD Procedure

Output	Description
status_inout	command return code; 0 for success and not 0 for failure

Index

A

ADD_ATTRIBUTE procedure, 2-11
add_clause function, 2-78
add_cond function, 2-82
ADD_CONTENT procedure, 2-12
ADD_EXTRA_HEADER procedure, 2-8
ADD_MIME_MARKER procedure, 2-9
APIs
  IM_API, 2-20
  IOSend, 2-2
  server side rules, 2-68
audience for this guide, x
AUTHENTICATE procedure, 2-5

B

BlindCopyTo function, 2-29
bundled interface
  GetMessageProp function, 2-45
  SetMessageProp function, 2-47

C

CloseFolder function, 2-40
CopyMessage function, 2-43
CopyToFolder function, 2-41
create_rule function, 2-80
CreateFolder function, 2-25

D

delete_rule function, 2-83
DeleteFolder function, 2-26
DeleteFrom function, 2-42
DeleteMessage function, 2-44
DeleteRule Function, 2-75
DESCRIBE_PART procedure, 2-9
DisabledLogging function, 2-24
DisableRule function, 2-74
documentation, xi

E

EnableLogging function, 2-24
EnableRule function, 2-73

F

FOLDER_CARBON_COPY procedure, 2-13
ForwardTo function, 2-30
ForwardWithTemplate function, 2-31

G

GetAttachmentBody function, 2-55
GetAttachmentData function, 2-56
GetExtendedHdrs function, 2-53
GetFolderExp function, 2-27
GetInlcusionID function, 2-58
GetMessageBody function, 2-54
GetMessageHdrs function, 2-52
GetMessageProp function, 2-45, 2-48
GetNewMail function, 2-35
GetNextAttachment function, 2-57
GetNextMessage function, 2-39
GetPartList function, 2-51

H

HasNewMail function, 2-34
hierarchical folder interface
  CopyToFolder function, 2-41
  DeleteFrom function, 2-42
  MoveToFolder function, 2-41

I

IM_API
  function reference, 2-21
  overview, 1-2
  sample script, 2-59
  using the IM_API, 2-20
installation
  documentation, xi
Internet Messaging
  documentation, xi
IOSend API
  function reference, 2-5
  overview, 1-2
  process overview, 2-4
  sample script, 2-14, 2-16
  using the IOSend API, 2-2

L

ListFolder function, 2-36
ListRules function, 2-73
ListTemplates function, 2-33

M

Messaging Folder functions, 2-40
MIME messages
  sending, 2-2
MoveMessage function, 2-44
MoveToFolder function, 2-41

N

non-bundled interface
  GetMessageProp function, 2-48
  SetMessageProp function, 2-49
non-hierarchical folder interface
  CopyMessage function, 2-43
  DeleteMessage function, 2-44
  MoveMessage function, 2-44

O

OpenFolder function
  array interface, 2-37
  iterative interface, 2-38
OpenMessageAttachment function, 2-58
OrderRules Function, 2-75

P

PL/SQL APIs
  descriptions, 1-2
  IM_API function reference, 2-21
  IM_API sample script, 2-59
  IOSend function reference, 2-5
  IOSend process overview, 2-4
  server side rules, 2-68
  server side rules sample script, 2-85
  using the IM_API, 2-20
preface, ix

R

reader's comment form, vii
release notes
  documentation, xi
RenameFolder function, 2-27
ReplyTo function, 2-31
ReplyWithTemplate function, 2-32

S

SaveRuleDetail function, 2-76
SendSimpleMessage function, 2-33
server side rules
  overview, 1-2
  sample script, 2-85
  using server side rules, 2-68
  viewing a list of rules, 2-84
SetFolderExp function, 2-28
SetMessageProp function, 2-47, 2-49

start_condition function, 2-77
SUBMIT_HEADER procedure, 2-6
SUBMIT_MESSAGE procedure, 2-14

T

toggle_rule function, 2-83
typographic conventions, x