The Mail Kit

The Device Kit − Table of Contents

The Mail Kit 1

The Mail Daemon 5

BMailMessage 10

The Mail Kit: Master Index 13

The Mail Kit

The Mail Kit provides Internet e−mail messaging services. These services include:

- Configuration of the user's mail accounts.
- Sending messages using the Simple Mail Transfer Protocol (SMTP).
- Receiving messages via the Post Office Protocol (POP).
- Automatic, timed sending and receiving of messages.
- Encoding and decoding base−64 encoded data.

An assortment of global C functions are provided by the Mail Kit to configure the mail daemon and process base−64 data. See "The Mail Daemon" for information on how to use these functions.

Outgoing mail messages are constructed and sent using the BMailMessage class.

Mail Message Files

Every mail message is stored in an individual file with attached attributes that describe the message in detail; you can query the file system to obtain information about the sender, subject, and receiver of the message, among other things. See "Querying Mail Messages" below for more detailed information and an example program.

Every message the user writes is saved in a file until it's sent by the mail daemon (and may or may not be deleted after being sent). Likewise, messages that the mail daemon has retrieved are also stored in files on a local disk.

The process of sending a mail message works something like this:

- The user writes a new mail message and chooses the mail writing program's "Send" option.
- The mail writing program creates a BMailMessage object and configures it based on the user's inputs by setting the "To," "Subject," and other header fields appropriately, and by storing the message content into the BMailMessage.
- The program then calls the BMailMessage object's **`Send()`** function to tell the mail daemon to send the message.
- The mail daemon creates a disk file that contains the message. The message content is stored in the file itself, and attributes are created to contain the "To," "Subject," and other relevant header fields. See "Querying Mail Messages" for more information. The message's status attribute is set to "New".
- The next time the mail daemon's **`check_for_mail()`** function is called (either automatically or explicitly), the daemon sends the message via SMTP, then changes the message's status attribute to "Sent". This is done for all mail messages whose status is "Pending".

After sending outgoing messages, the mail daemon will also check to see if any incoming mail is waiting to be retrieved. If there is, it proceeds something like this:

- The mail daemon fetches the first message from the mail server via POP.
- The daemon creates a new mail message file, and the message is written into the file. The file contains the Internet headers, message content, and all enclosures (if any).
- The mail daemon scans the message and adds attributes to the message file for each of the header fields, as well as a couple of extra attributes. These are described in detail in "Querying Mail Messages" below.
- The daemon continues reading mail messages from the mail server until there aren't any left.

Querying Mail Messages

The Mail Kit takes full advantage of the BeOS attribute and query system. Each message received is parsed by the mail daemon and important information about it is converted into a defined collection of attributes attached to the file. This makes it extremely easy to create applications that search for messages meeting specific parameters. In an example to follow shortly, we'll create a program that lists all unread messages.

Once the mail daemon has received a message and saved it to disk, any application can query the file system to locate messages that meet certain parameters, then read the attributes and message content to present information about that message to the user.

The following attributes are provided by the Mail Kit:

B_MAIL_ATTR_NAME	MAIL:name	Name of the mail file.
`B_MAIL_ATTR_STATUS`	MAIL:status	Message status.
`B_MAIL_ATTR_PRIOIRITY`	MAIL:priority	"Priority" field value.
`B_MAIL_ATTR_TO`	MAIL:to	"To" field value.
`B_MAIL_ATTR_CC`	MAIL:cc	"Cc" field value.

B_MAIL_ATTR_FROM	MAIL:from	"From" field value.
B_MAIL_ATTR_SUBJECT	MAIL:subject	"Subject" field value.
B_MAIL_ATTR_REPLY	MAIL:reply	"Reply−to" field value.
B_MAIL_ATTR_WHEN	MAIL:when	"When" field value.
B_MAIL_ATTR_FLAGS	MAIL:flags	Message flags.
B_MAIL_ATTR_RECIPIENTS	MAIL:recipients	List of message recipients.
B_MAIL_ATTR_MIME	MAIL:mime	The MIME version used.
B_MAIL_ATTR_HEADER	MAIL:header_length	Length of the message header.
B_MAIL_ATTR_CONTENT	MAIL:content_length	Length of the message content.

B_MAIL_ATTR_NAME is a string that identifies the name of the sender.

B_MAIL_ATTR_STATUS is a string that identifies the status of the message. Possible values are:

"Error"	An error occurred trying to send the message.
"New"	The message has not been read yet.
"Pending"	The message has not been sent yet.
"Read"	The message has been read.
"Sent"	The message has been sent.

B_MAIL_ATTR_PRIORITY is a string that contains the value of the Priority field in the message.

B_MAIL_ATTR_TO, **B_MAIL_ATTR_FROM**, and **B_MAIL_ATTR_REPLY** are strings that contain the primary recipient's e−mail address, the sender's e−mail address, and the sender's reply−to address.

B_MAIL_ATTR_CC contains the addresses to whom the message is carbon copied.

B_MAIL_ATTR_SUBJECT is a string containing the subject of the message.

B_MAIL_ATTR_WHEN is a BeOS time field (**B_TIME_TYPE**) containing the message's "When" field.

B_MAIL_ATTR_FLAGS contains 32−bit integer (int32) flags which can be any combination of the following values:

B_MAIL_PENDING	Message is waiting to be sent.
B_MAIL_SENT	Message has been sent.
B_MAIL_SAVE	Mail will be saved after being sent.

The **B_MAIL_ATTR_FLAGS** attribute is BeOS−specific.

B_MAIL_ATTR_RECIPIENTS contains a list of all recipients of the message (primary, cc, and bcc), but is only valid for outgoing messages (this attribute doesn't exist on incoming messages).

B_MAIL_ATTR_MIME contains a string that defines the version number of the MIME specification used to transmit any enclosures attached to the file. This attribute is only present if the file has one or more enclosures.

B_MAIL_ATTR_HEADER and **B_MAIL_ATTR_CONTENT** contain the lengths, in bytes, of the header and content portions of the message. They're both **int32** type data.

The following attributes are indexed:

B_MAIL_ATTR_NAME

B_MAIL_ATTR_STATUS
B_MAIL_ATTR_PRIORITY

B_MAIL_ATTR_TO
B_MAIL_ATTR_CC
B_MAIL_ATTR_FROM
B_MAIL_ATTR_SUBJECT

B_MAIL_ATTR_REPLY
B_MAIL_ATTR_WHEN
B_MAIL_ATTR_FLAGS

The headers, the contents of the messag,e and the enclosures (in base−64 encoded form) can all be found in the file itself and can be read using a BFile object. See "The Storage Kit" in the for further information on reading files.

Queries and Mail Messages

Now that you know what attributes are available on mail message files, and which attributes are indexed, you can consider all the clever things you can use them for. Let's look at an example program that, from a Terminal window, lets you see a list of the unread mail you have.

```
void main(void) {
   BQuery query;
   BNode node;
   BVolume vol;
   BVolumeRoster vroster;
   entry_ref ref;
   char buf[256];
   int32 message_count = 0;

   vroster.GetBootVolume(&vol);
   query.SetVolume(&vol);
```

The program begins by establishing needed variables, then using a BVolumeRoster to set the query's search volume to the boot disk. This is covered in more detail in the Storage Kit chapter.

```
   if (query.SetPredicate("MAIL:status = New") != B_OK) {
      printf("Error: can>t set query predicate.n");
      return;
   }
```

Then the query is configured to search for new mail. New messages can be identified by the **B_MAIL_ATTR_STATUS** attribute (called "MAIL:status") having a string value of "New". If an error occurs, the program prints an error end returns.

```
   if (query.Fetch() != B_OK) {
      printf("Error: new mail query failed.n");
      return;
   }
```

The query is told to fetch. Again, if this fails, an error message is displayed and the program returns.

```
   while (query.GetNextRef(&ref) == B_OK) {
      message_count++;      // Increment message counter
```

The loop scanning through the fetched new messages begins by incrementing the counter of new messages received.

```
      if (node.SetTo(&ref) != B_OK) {
         printf("Error: error scanning new messages.n");
         return;
      }
```

Then a BNode is set to reference the message file. If this fails, the program displays an error message and quits.

```
      buf[0] = >0>;      // If error, use empty string
      node.ReadAttr(B_MAIL_ATTR_FROM, B_STRING_TYPE, 0, buf, 255);
      buf[20] = >0>;      // Truncate to 20 characters
      printf("%3d From: %−20s", message_count, buf);
```

The buffer we're using the receive the attribute values is initialized to an empty string, then we call BNode's **ReadAttr()** function to read the **B_MAIL_ATTR_FROM** attribute into the buffer. We then truncate the read string to 20 characters for display purposes (to make it fit into the table we're outputting) and print the message number and sender information.

```
      buf[0] = >0>;      // If error, use empty string
      node.ReadAttr(B_MAIL_ATTR_SUBJECT, B_STRING_TYPE, 0, buf, 255);
      buf[40] = >0>;      // Truncate to 40 characters
      printf("   Sub: %sn", buf);
   }
```

The buffer is reset to an empty string and the **B_MAIL_ATTR_SUBJECT** attribute is read into it. This string is truncated to 40 characters, then printed.

This loop continues until no more new messages are found; this is detected when **GetNextRef()** returns an error.

```
   if (message_count) {
      printf("%d new messages.n", message_count);
   }
   else {
      printf("No new messages.n");
   }
```

```
}
```

Finally, the number of new messages is printed. If there aren't any messages, we very politely print "No new messages." rather than "0 new messages," for a little added panache.

This simple example demonstrates how you can use the attributes provided by the Mail Kit to create mail message reading applications. The message body and attachments are stored within the file itself, and can be read using the functions described in the BFile section in the Storage Kit chapter of the "Be Developer's Guide."

The E−mail File Type

The mail daemon ensures, when it's first launched at system boot time, that the BeOS File Type database includes an entry for e−mail files. E−mail files have the MIME string "text/x−email", which is represented by the constant **B_MAIL_TYPE**.

The E−mail entry in the File Type database includes a list of the attributes on which you can search. You can look up this information using the BMimeType class's **GetAttrInfo()** function:

```
BMimeType mime;
BMessage message;

mime.SetTo(B_MAIL_TYPE);
mime.GetAttrInfo(&message);
```

After running this code, the *message* contains the description of the attributes available on e−mail files. The message has three useful arrays of items:

"attr:public_name"	The user−readable name of the attribute.
"attr:name"	The attribute name used for BNode and **fs_attr_*** calls.
"attr:type"	The type of data the attribute contains.

You can examine the items in each of these arrays in the message to get useful information about the attributes you can reference on the e−mail files. For example:

```
printf("Public name: %sn", FindString("attr:public_name", 1));
printf("Name: %sn", FindString("attr:name", 1));
```

This code will print the public name and the attribute name of the second attribute registered in the File Type database entry for e−mail files. In the current implementation of the Mail Kit, this would print:

```
Public name: Subject
Name: MAIL:subject
```

The number of (and order of) attributes in the database entry might change in the futureand that's where the File Type database comes in handy. If, a year from now, Be adds more attributes to e−mail files, your File Type database−savvy application won't have to be updated to support them.

At this time, the File Type database has attribute information records for each of the following attributes:

B_MAIL_ATTR_NAME
B_MAIL_ATTR_STATUS
B_MAIL_ATTR_PRIORITY

B_MAIL_ATTR_TO
B_MAIL_ATTR_FROM
B_MAIL_ATTR_SUBJECT

B_MAIL_ATTR_REPLY
B_MAIL_ATTR_WHEN

You can obtain information on these attributes, their formats, and their user−readable names by looping through the arrays in the message until the BMessage:**Find...()** function returns **NULL** or **B_BAD_INDEX**.

For more information on the BMimeType class and the File Type database, see BMimeType in The Storage Kit.

The Mail Daemon

Derived from: none

Declared in: be/kit/mail/E−mail.h

Library: libmail.so

Every computer running the BeOS has a **mail daemon**; this is a local process that's responsible for sending e−mail to and receiving e−mail from a mail server. The mail server that the daemon talks to is a networking application that's either part of your Internet Service Provider's services, or that's running on a local "mail repository" machine.

The functions described in this section tell you how to manage the mail daemon's connection with the mail serverhow to tell the daemon which mail server to communicate with, how to tell the mail daemon to send and retrieve e−mail, how to automate mail retrieval, and so forth.

Many of the functions described here are user−accessible through the E−mail preference application. These functions should generally not be used; the settings they control belong to the user, and your application should usually avoid changing the user's settings. The only legitimate reason to use these configuration setting functions is if you want to build your own E−mail preference application.

The other functions, such as **`forward_mail()`**, **`check_for_mail()`**, **`encode_base64()`**, and **`decode_base64()`**, might be legitimately used by your e−mail program.

The architecture of an e−mail message isn't discussed here; for that information, see "Mail Messages (BMailMessage)."

The Mail Daemon and the Mail Server

The mail daemon can talk to two different kinds of mail server:

- The **Post Office Protocol** ("POP") server manages individual mail accounts. When the BeOS mail daemon wants to retrieve mail that's been sent to a user, it must tell the mail server which POP accouant it's retrieving mail for.
- The **Simple Mail Transfer Protocol** ("SMTP") server manages mail that's being sent out onto the network. Messages sent through an SMTP server will eventually find their way to a POP server to be received by the destination user.

The POP and the SMTP servers are identified by their hosts' names (in other words, the names of the machines on which the servers are running). The mail daemon can only talk to one POP and one SMTP server at a time, but can talk to the two of them simultaneously. Usuallybut not alwaysthe POP and SMTP servers reside on the same machine, and so are identified by the same name.

To set the identities of the POP host, you fill in the fields of a **`mail_pop_account`** structure and pass the structure to the **`set_pop_account()`** function. As the name of the structure implies, **`mail_pop_account`** encodes more than just the names of the server's host. It also identifies a specific user's POP mail account; the complete definition of the structure is this:

```
typedef struct
{
   char pop_name[B_MAX_USER_NAME_LENGTH];
   char pop_password[B_MAX_USER_NAME_LENGTH];
   char pop_host[B_MAX_HOST_NAME_LENGTH];
   char real_name[128];
   char reply_to[128];
   int32 days;
   int32 interval;
   int32 begin_time;
   int32 end_time;
} mail_pop_account;
```

The **`pop_name`**, **`pop_password`**, and **`pop_host`** fields in the **`mail_pop_account`** structure represent the username, password, and POP server host of the e−mail user. The **`real_name`** is the user's real name, and **`reply_to`** is the e−mail address to which replies should be sent.

The **`days`** field can contain any of the following flags to specify which days of the week the mail daemon should automatically check mail for the described account:

`B_CHECK_NEVER`	Don't automatically check the account's mail.
`B_CHECK_WEEKDAYS`	Check the mail only on weekdays.
`B_CHECK_DAILY`	Check the mail every day.
`B_CHECK_CONTINUOUSLY`	Check continuously every day.

The **`interval`** specifies how many seconds apart each e−mail retrieval should be, and the **`begin_time`** and **`end_time`** specify the time of day (in seconds) that automatic retrieval should begin and end.

The SMTP server can be selected by calling **`set_smtp_host()`**, passing in a pointer to the SMTP host's name.

Sending and Retrieving Mail

Messages that are retrieved (from the mail server) by the mail daemon are stored as individual files on the user's hard disk, from whence they are plucked and displayed by a mail−reading application (a "mail reader"; Be supplies a simple mail reader called BeMail). Similarly, messages that the user composes (in a mail composition application) and sends are stored as individual files until the mail daemon comes along and passes them on to the mail server.

Sending and retrieving mail is the mail daemon's most important function. Both actions (server−to−database and database−to−server transmission) are performed through the **check_for_mail()** function.

The BMailMessage class provides a convenient means for creating and sending new mail messages; visit the section on that class for further information and a simple example.

Mail that has been retrieved by the mail daemon can be identified and queried using the mail attributes defined by the Mail Kit. By using the BQuery class, you can scan all newly−received mail messages and parse the message file to present each message to the user. For a more in−depth discussion of the mail attributes and how to use them to your benefit, read "Querying Mail Messages."

Other Mail Daemon Features

The other mail structures and functions define the other features that are provided by the mail daemon. These features are:

- **Mail notification**. The **mail_notification** structure (passed through the **set_mail_notification()** function) lets you tell the daemon how you would like it to tap you on the shoulder when it has new mail for you to read. Would you like it to display an alert panel? Squawk at you? Both? This can be configured by the user in the E−mail preference application.
- **Mail forwarding**. The **forward_mail()** function lets you ask the Mail Kit to forward a message to one or more other accounts.
- **Base−64 encoding and decoding**. The **encode_base64()** and **decode_base64()** functions let you easily handle ASCII−encoded file attachments.

Functions

check_for_mail()

```
status_t check_for_mail(int32 *incoming_count = NULL)
```

Sends and retrieves mail. More specifically, this function asks the mail daemon to retrieve incoming messages from the POP server and send any queued outgoing messages to the SMTP server. The number of POP messages that were retrieved are stored in the variable pointed to by *incoming_count*. If you specify **NULL** for *incoming_count*, **check_for_mail()** won't return the number of messages retrieved. You should specify **NULL** unless you really want to know how many messages were retrieved, since requesting this information could potentially slow down the retrieval process.

If all is well in the mail world, this function returns **B_OK**; otherwise, it returns a highly useful result code.

RETURN CODES

- **B_OK**. Mail was sent and retrieved without incident.
- **B_MAIL_NO_DAEMON**. The mail daemon isn't running.
- **B_MAIL_UNKNOWN_USER**. The POP server doesn't recognize the user name.
- **B_MAIL_WRONG_PASSWORD**. The POP server doesn't recognize the password.
- **B_MAIL_UNKNOWN_HOST**. The POP or SMTP server can't be found.
- **B_MAIL_ACCESS_ERROR**. The connection to the POP or SMTP server failed.

count_pop_accounts()

```
int32 count_pop_accounts(void)
```

Returns the number of POP accounts that have been configured.

The mail daemon currently supports only one POP account, so this function will always return 1. You shouldn't assume there will only be one POP account, though, as this will probably change in the future.

decode_base64()

ssize_t **decode_base64(**char *_out_, char *_in_, off_t _length_, bool _replace_cr = false_**)**

Decodes the base−64 data pointed to by _in_, which is _length_ bytes long, and writes the decoded output into the buffer pointed to by _out_. If _replace_cr_ is **true**, carriage return characters in the output are converted into newlines, otherwise the data is returned in its original, unaltered, form.

You would typically specify _replace_cr_ as **true** if you're decoding an ASCII text document, and as **false** if decoding a binary file.

This function returns the size of the output data that's been stored in the _out_ buffer.

You must be certain, in advance, that the output buffer is large enough to hold the decoded data, or this function will do bad things.

encode_base64()

ssize_t **encode_base64**(char *_out_, char *_in_, off_t _length_)

Encodes the data pointed to by _in_, which is _length_ bytes long, and writes the base−64 encoded output into the buffer pointed to by _out_.

This function returns the size of the output data that's been stored in the _out_ buffer.

You must be certain, in advance, that the output buffer is large enough to hold the encoded data, or this function will do bad things.

forward_mail()

status_t **forward_mail**(entry_ref *_message_ref_, const char *_recipients_,
bool _now = true_)

Forwards the mail message specified by _message_ref_ to the list of users given by _recipients_. The list of user names specified in _recipients_ must be separated by commas and/or whitespace, and must be null−terminated.

If the _now_ parameter is **true**, the messages will be sent immediately; if **false**, the message will be queued up to be sent the next time **check_for_mail()** is called, or the next time the mail daemon performs an automatic mail check.

RETURN CODES

- **B_OK**. The message was forwarded without error.
- **B_MAIL_NO_RECIPIENT**. No valid recipients were specified.
- Errors returned by **send_queued_mail()**, if _now_ is **true**.

get_mail_notification() , set_mail_notification()

status_t **get_mail_notification**(mail_notification *_notification_settings_)

status_t **set_mail_notification**(mail_notification *_notification_settings_,
bool _save = true_)

get_mail_notification() fills the specified **mail_notification** structure with information describing how the user is currently being notified of received e−mail. There are two possible notification signals: the mail alert panel and the system beep. The **mail_notification** structure looks like this:

```
typedef struct
{
   bool alert;
```

```
   bool beep;
} mail_notification;
```

get_mail_notification() always returns **B_OK**. If the current settings can't be checked (for example, if the user has never configured mail), *alert* will be returned as the default value of **false**, and *beep* will be **true**.

set_mail_notification() accepts a pointer to a **mail_notification** structure and configures the system to report incoming mail using the methods specified therein. If the *save* argument is **true**, the change is set as the new default and will be remembered when the computer is shut down. If **false**, the change is temporary.

RETURN CODES

- **B_OK**. The notification was successfully set or retrieved.
- **B_NO_REPLY**. The mail daemon didn't respond to the request.

---

get_pop_account() , set_pop_account()

status_t **get_pop_account**(mail_pop_account *_account_info_, int32 *_index = 0_)

status_t **set_pop_account**(mail_pop_account *_account_info_, int32 *_index = 0_)

Get and set the specified POP account's information. The **mail_pop_account** structure is defined as follows:

```
typedef struct
{
   char pop_name[B_MAX_USER_NAME_LENGTH];
   char pop_password[B_MAX_USER_NAME_LENGTH];
   char pop_host[B_MAX_HOST_NAME_LENGTH];
   char real_name[128];
   char reply_to[128];
   int32 days;
   int32 interval;
   int32 begin_time;
   int32 end_time;
} mail_pop_account;
```

The **pop_name**, **pop_password**, and **pop_host** fields in the **mail_pop_account** structure represent the username, password, and POP server host of the e−mail user. The **real_name** is the user's real name, and **reply_to** is the e−mail address to which replies should be sent.

The **days** field can contain any of the following flags to specify which days of the week the mail daemon should automatically check mail for the described account:

B_CHECK_NEVER	Don't automatically check the account's mail.
B_CHECK_WEEKDAYS	Check the mail only on weekdays.
B_CHECK_DAILY	Check the mail every day.
B_CHECK_CONTINUOUSLY	Check continuously every day.

The **interval** specifies how many seconds apart each e−mail retrieval should be, and the **begin_time** and **end_time** specify the time of day (in seconds) that automatic retrieval should begin and end. If **begin_time** and **end_time** are the same, the daemon checks mail round−the−clock.

Eventually these functions will support multiple POP accounts; at this time, the Mail Kit only supports one POP account, so you must use an *index* of 0. Any other index will result in a **B_BAD_INDEX** error.

get_pop_account() fills the specified **mail_pop_account** structure with the information on the POP account, and **set_pop_account()** takes the information in the buffer and saves it as the new default.

RETURN CODES

- **B_OK**. The notification was successfully set or retrieved.
- **B_BAD_INDEX**. An index other than 0 was specified.
- **B_NO_REPLY**. The mail daemon didn't reply to the request.

---

get_smtp_host() , set_smtp_host()

status_t **get_smtp_host**(char *_smtp_host_)

status_t **set_smtp_host**(char *_smtp_host_, bool _save_ = true)

get_smtp_host() returns in the buffer pointed to by _smtp_host_ the name of the SMTP host as currently configured. The buffer should be at lest **B_MAX_HOST_NAME_LENGTH** bytes long.

set_smtp_host() sets the SMTP host through which mail will be sent in the future to the specified host. If _save_ is **true**, the new setting becomes the default and will persist through a reboot of the computer; otherwise, the change is only temporary.

RETURN CODES

- **B_OK**. The notification was successfully set or retrieved.
- **B_NO_REPLY**. The mail daemon didn't respond to the request.

---

send_queued_mail()

status_t **send_queued_mail(**void**)**

Tells the mail daemon to send all pending outgoing mail.

RETURN CODES

- **B_OK**. Mail transfer intitiated successfully.
- Errors from BMessage::**SendMessage()**.

BMailMessage

Derived from: none

Declared in: be/kit/mail/E−mail.h

Library: libmail.so

Summary

The BMailMessage class provides an easy way to send e−mail messages. If you want to do it the hard way, look up the SMTP RFC and start plodding your way through the Network Kit documentation. You'll get it working one of these days.

Or you can sail right on through and be sending e−mail from your own applications in a matter of minutes using your friend, the BMailMessage.

Constructing a Mail Message

To send an e−mail, you simply construct a new BMailMessage object, add a "To" header field, add the content, and send the message on its way. For example:

```
BMailMessage *mail;
char *message;

mail = new BMailMessage();
mail->AddHeaderField(B_MAIL_TO, "bob@uncle.com");
mail->AddHeaderField(B_MAIL_SUBJECT, "Hi");
message = "Hi, Uncle Bob!";
mail->AddContent(message, strlen(message));
```

This is a pretty basic message. The subject is "Hi," the message is sent to "bob@uncle.com," and the message body is "Hi, Uncle Bob!"

You can add other fields, including carbon−copy (CC) and blind−carbon−copy (BCC) fields, and you can add attachments. For example, if you want to also attach a file called "/boot/home/file.zip," you can do the following:

```
mail->AddEnclosure("/boot/home/file.zip");
```

Once your message has been constructed, you can send it by calling **Send()**:

```
mail->Send();
```

That's the basic technique behind sending e−mail under the BeOS. The mail daemon also fetches incoming mail from a POP server, but you can't use the BMailMessage class to read these messages; you use the BeOS BQuery and BNode classes to locate messages of interest and obtain information about them. See "Querying Mail Messages" for more information.

Constructor and Destructor

BMailMessage()

```
BMailMessage(void)
```

Creates and returns a new BMailMessage object, which is empty. You need to call other functions defined by this class to fill out the message.

~BMailMessage()

```
~BMailMessage()
```

Destroys the BMailMessage, even if the object's fields are "dirty." For example, if you create a new BMailMessage object with the intention of sending a message, fill out some or all of the fields, and then delete the object, the object is destroyed without being sent.

Member Functions

AddContent()

```
status_t AddContent(const char *text, int32 length,
    uint32 encoding = B_ISO1_CONVERSION, bool replace = false)
```

status_t **AddContent**(const char *_text_, int32 _length_, const char *_encoding_, bool _replace_ = **false**)

Adds the specified _text_ (which contains _length_ characters) to the BMailMessage object's content. The text's encoding is specified by the _encoding_ parameter, either directly or by pointer.

If _replace_ is **true**, any existing text is deleted before the new content is added; otherwise, the specified text is appended to the end of the existing message content.

RETURN CODES

- **B_OK**. The content was changed without error.
- **B_ERROR**. Unable to add the new content.

AddEnclosure()

status_t **AddEnclosure(**entry_ref *_enclosure_ref_, bool _replace_ = _false_**)**

status_t **AddEnclosure**(const char *_path_, bool _replace_ = _false_**)**

status_t **AddEnclosure**(const char *_mime_type_, void *_data_, int32 _length_, bool _replace_ = _false_**)**

Adds an attachment to the message. The first two forms of **AddEnclosure()** add a file to the message, given either an **entry_ref** pointer or a pathname. The third form adds a block of memory (of the given _length_) to the message as an enclosure, with the specified MIME type.

If _replace_ is **true**, any existing attachmentsincluding the body of the messageare removed before the new one is added; otherwise the new enclosure is added, leaving previous attachments intact.

If you specify **true** for _replace_, not only will all existing enclosures be discarded, but so will the content of the message body itself.

RETURN CODES

- **B_OK**. The content was changed without error.
- **B_ERROR**. Unable to add the new enclosure.

AddHeaderField()

status_t **AddHeaderField(**const char *_field_name_, const char *_field_str_, bool _replace_ = _false_**)**

Adds a header field to the BMailMessage object. The value of the field whose name is specified by _field_name_ is set to the string specified by _field_str_.

If _replace_ is **true**, all existing header fields of the specified name are deleted before adding the new header field; if _replace_ is **false**, a new header whose field is named _field_name_ is added.

RETURN CODES

- **B_OK**. The content was changed without error.
- **B_ERROR**. Unable to add the new header field.

Send()

status_t **Send(**bool _send_now_ = **false**, bool _remove_when_sent_ = _false_**)**

Queues the message for transmission. If _send_now_ is **true**, the message is sent immediately; otherwise, it is placed in the queue to be sent the next time **check_for_mail()** is called or the mail daemon performs an automatic mail check.

If the _remove_when_sent_ argument is **true**, the message will be deleted from the user's disk drive after it has been sent; otherwise, it will be saved

for posterity.

RETURN CODES

- **`B_OK`**. The content was changed without error.
- **`B_MAIL_NO_RECIPIENT`**. There needs to be either a "To" or "Bcc" field in the message.
- Errors from **`BMessage::SendMessage()`**.

The Mail Kit: Master Index

A

AddEnclosure()	BMailMessage
AddHeaderField()	BMailMessage

C

Constructing a Mail Message	BMailMessage
Constructor and Destructor	BMailMessage
count_pop_accounts()	The Mail Daemon

D

E

encode_base64()	The Mail Daemon

F

Function Summary	BMailMessage
Functions	The Mail Daemon

G

get_pop_account()	The Mail Daemon
get_smtp_host()	The Mail Daemon

M

The Mail Daemon	The Mail Daemon
The Mail Daemon	The Mail Daemon
The Mail Kit	The Mail Kit
The Mail Kit	The Mail Kit
BMailMessage	BMailMessage
BMailMessage()	BMailMessage
~BMailMessage()	BMailMessage
Mail Message Files	The Mail Kit

Member Functions	BMailMessage

O

Q

Querying Mail Messages	The Mail Kit

S

send_queued_mail()	The Mail Daemon
Sending and Retrieving Mail	The Mail Daemon
set_mail_notification()	The Mail Daemon
set_pop_account()	The Mail Daemon
set_smtp_host()	The Mail Daemon