Messaging Server Plug-in API Guide

Messaging Server Plug-ins

Communicator 4.0, January 1998

Recycled and Recyclable Paper

Version 4.0

©Netscape Communications Corporation 1998
All Rights Reserved

Printed in USA
99 98 97 10 9 8 7 6 5 4 3 2 1

Netscape Communications Corporation 501 East Middlefield Road, Mountain View, CA 94043

Contents

About This Guide

The *Messaging Server Plug-in API Guide* describes the programming interface (API) to Netscape Messaging Server 3.0. It describes the Messaging Server plug-in interface and support routines.

The *Messaging Server Plug-in API Guide* is intended for developers who want to extend the functionality of Netscape Messaging Server for site-specific reasons.

This guide assumes that you are using the Messaging Server Plug-in API 3.0. This API is available on the Solaris, HP-UX, IRIX, and Windows NT platforms.

This chapter includes the following sections:

- Conventions Used in This Guide
- Where to Find Messaging Server Plug-in Files
- Where to Find Developer Information

Conventions Used in This Guide

All program code listings, URLs, and other program names appear in Courier, a monospace font.

This guide emphasizes information with several types of note formats:

Note Information of interest to the developer but not essential to understanding the surrounding topic. §

Warning Information that can affect the development decisions you make or the development environment you choose. Don't miss these notes. §

Where to Find Messaging Server Plug-in Files

You can find the `plugin.h` header file at `http://help.netscape.com/products/server/messaging/3x/info/apiheader.html`.

Where to Find Developer Information

For Netscape developer information, see the Netscape DevEdge site.

For more information about server APIs, see these Netscape documents:

- *The Directory SDK Programming Guide*
- *Collabra Server API Guide*

For details about messaging elements, refer to the following RFCs (Request for Comments):

- RFC 821: "Simple Mail Transfer Protocol"
- RFC 822: "Standard for the Format of ARPA Internet Text Messages"

1

Introduction

A Messaging Server Plug-in is a separate code module that behaves as though it is a part of the Netscape Messaging Server. You can use the Messaging Server Plug-in API to create plug-ins that add message-handling functionality to the Messaging Server. The plug-ins you write become part of the server address space and are called automatically whenever a new message comes in.

This chapter explains what you can and cannot do with the Messaging Server Plug-in API, provides some basic definitions, and lists the stages involved in message processing.

- How You Can Use the Messaging Server Plug-in API
- Messages and Message Processing

How You Can Use the Messaging Server Plug-in API

The Netscape Messaging Server 3.0 plug-in interface allows third party developers to plug in site-specific functionality. Here's what you can do with the Messaging Server Plug-in API:

- Process the RFC 822 header and body at exposed module boundaries. You can rewrite these elements in protocol-compliant format before the message undergoes further processing in Netscape Messaging Server.

Customers can use this capability to perform functions that change the header or body of the messages or both. These functions can include character-set conversions and content filters such as HTML-to-text and PostScript-to-text convertors. These filters can implement site-specific firewall functionality to filter out suspicious attachments, either outgoing or incoming.

- Split the message. You can split the envelope recipients and change the envelope sender. This is useful when the plug-in module needs to alter the content of the message for a subset of the original recipients.

Here's what you cannot do with the plug-in API:

- Override the internal functionality of the Messaging Server.
- Change the sequence of modules in Messaging Server processing.

Note The Messaging Server Plug-in API is available on the Solaris, HP-UX, IRIX, and Windows NT platforms.§

Messages and Message Processing

The Messaging Server Plug-in API uses a message format that conforms to the definition in RFC 822: "Standard for the Format of ARPA Internet Text Messages." A single message is made up of control, header, and body files.

- Control files contain the envelope of the message.
- Header files contain the RFC 822 headers of the message.
- Body files contain the RFC 822 body of the message.

To process a message, an application goes through a sequence of steps. Each step is implemented by an executable program called a module, which a dispatcher program spawns in sequence to carry out message processing tasks. The sequence of tasks varies depending on the message type.

The control, header, and body files of the message make up the key input to the module. The module can do one of two things. It can either use this input, along with the user and module databases, to generate one or more messages for further processing, or it can pass the responsibility for the message on to another Message Transfer Agent (MTA).

Finally, it delivers the message to the mailbox of one of the users or bounces it. For sample code that illustrates each step of this process, see "Messaging Server Sample Plug-in."

Chapter

2

Using the Messaging Server Plug-in API

This chapter contains basic information about the elements and functions of the Messaging Server Plug-in API.

This chapter introduces the functions and result codes of the plug-in (external) code and discusses the functions provided by Messaging Server to manipulate its fundamental data structures. It concludes with platform-specific information for generating the shared object.

Plug-in Entry Points

With the Messaging Server Plug-in API, you can plug in additional functionality at two stages in message processing:

- `PostSmtpAccept`: This stage occurs immediately after the message is received off the wire. At this stage neither the RFC 822 portion (header and body) nor the envelope is touched by the Messaging Server, and the message may not be fully standards compliant.
- `PreSmtpDelivery`: This stage occurs just before the message is handed off to another host.

The envelope information for the message is maintained in memory until the plug-in entry point returns to the server. When this happens, it is written atomically to the disk. If several entry points for a single stage are cascaded, the server waits to write the envelope information after all the entry points return. The server writes the entry points in the order they appear in the configuration file.

Use this format to specify plug-in functionality in the plug-in configuration file:

```
Stage SharedObject funcs=comma-separated list of functions [name=value,...]
```

The `Stage` parameter represents one of the stages in message processing, either `PostSmtpAccept` (immediately after the message is received off the wire) or `PreSmtpDelivery` (just before the message is handed off to another host).

The `SharedObject` is the dynamically loadable code that implements the additional functionality. For more infomation, se "Generating the Shared Object."

The `funcs` parameter is made up of a comma-separated list of functions that implement the additional functionality.

The `name=value` parameter represents arbitrary name-value pairs that are passed to the function to add more functionality. For example, you could set a path for a temporary file such as "`c = /temp`." To get the value part of one of these pairs, use `pblock_findval`.

For example, a plug-in configuration file could contain lines like these:

Unix

```
PostSmtpAccept /foo/bar/libxlate.so funcs=decode-euc
PostSmtpDelivery /foo/bar/libxlate.so funcs=encode-iso no-xlate-domains="*.kr"
```

Windows NT

```
PostSmtpAccept d:\plugins\libxlate.dll funcs=decode-euc
PostSmtpDelivery d:\plugins\libxlate.dll funcs=encode-iso no-xlate-domains="*.kr"
```

In the first lines of both examples, the plug-in selects the `PostSmtpAcceptmessage` processing stage, and gives the full path of the `libxlate.so` shared object as the dynamically loadable code. The shared object location is indicated differently for Unix and Windows NT. The `funcs` list tells which function, `decode-euc`, implements the additional functionality.

In the second lines of both examples, the plug-in selects the `PostSmtpDelivery` processing stage, and gives the full path of the `libxlate.so` shared object. The `funcs` list includes a different function, `encode-iso`. In this line, the `name=value` parameter contains `no-xlate-domains="*.kr`."

For example, a plug-in configuration file could contain lines like these:

Plug-in Function Format and Result Codes

The plug-in function has this format:

```
int function (pblock *Config,
              Message **InMessage,
              Message ***OutMessage);
```

The `Config` parameter represents a hash table organization of name-value pairs. The hash table contains all the optional parameters on the configuration command line in `name=value` format. For example, the `no-xlate-domains` parameter in the example in "Plug-in Entry Points" is passed in the parameter block `Config`.

The `InMessage` parameter is an input-only parameter that represents a pointer to an array of pointers to structures of type `Message`, terminated by a `null` pointer. Every pointer to the `Message` structure represents a separate message. The simplest and perhaps the most common case is to have only one pointer in this array.

The `OutMessage` parameter represents the result of the execution of the site-specific plug-in code. This parameter is required only if the plug-in code generates more than one message from a single input message. The plug-in code should return `null` in the `OutMessage` parameter in these cases:

- The plug-in code detects that no part of the message headers or body must be rewritten.

- The message should be rewritten for all recipients, so the message need not be split.

If you split the message into multiple messages, do so immediately after it is received by the Messaging Server, at the `PostSmtpAccept` stage. If the plug-in code does not split the input message (for example, if it encodes the message for all the recipients), the `OutMessage` output parameter is not necessary.

The plug-in code should return one of these codes to the Messaging Server. The Messaging Server Plug-in API return messages are listed in "Result Codes."

- `MSG_CONTINUE`. The plug-in has performed its function. The resulting messages should go on to the Mail Server for further processing. This result is possible from any stage in Mail Server operation.
- `MSG_NOACTION`. The plug-in module did not do anything. Proceed with further message processing.

To see how the plug-in function is used in a sample plug-in DLL, see "Messaging Server Sample Plug-in." For more information about stages in message processing, see "Messages and Message Processing." For information about the `Message` structure, see "Result Codes."

You can use Messaging Server Plug-in API functions to allocate and manipulate `Message` structures. For more information, see "Using Messaging Server Plug-in Functions."

Data Structures

Message Server Plug-in functions use two fundamental data structures, the `pblock` parameter block and the `Message` structure. To help you use these structures, this section provides a general description of their characteristics and contents.

- Parameter Block
- Message Structure

Warning The type definitions in this section are for informational purposes only. They do not reproduce the actual definitions of the structures. All types mentioned in this guide are opaque, and members of the type defined structures are not exposed in the header files. The only access to the members is through the Messaging Server API. §

Parameter Block

The first fundamental data structure in the server code is the parameter block, `pblock`. The parameter block is a hash table keyed on the name string, which maps the name strings to their value character strings. A parameter block could be constructed as in this example:

```
typedef struct {
   char *name, *value; }
   pb_param;

struct {
   pb_param *param;
   struct pb_entry *next; };
   pb_entry;

typedef struct {
   int hsize;
   struct pb_entry **ht; }
   pblock;
```

In this example, the `pb_param` structure is used to manage name-value pairs, and `pb_entry` is a structure used to create linked lists of `pb_param` structures.

The parameter block in the example, `pblock`, is the hash table that holds `pb_entry` structures. Its contents are transparent to most code. The hash function is subject to change and is not made known to application functions.

To access this structure, use the `pblock_findval` function.

Warning This type definition is for informational purposes only. It does not reproduce the actual definition of the structure. §

Message Structure

The `Message` structure stores all attributes that define a single message. This includes envelope information, such as envelope sender and recipients, sender and per-recipient extensions, and so on, as well as RFC 822 attributes such as pointers to RFC 822 headers and RFC 822 body.

The `Message` structure could be constructed as in this example:

```
typedef char *N821Address;

typedef char *SmtpExt;

typedef struct { long magic; /* definition of Address structure */
   N821Address Addr;
   SmtpExt Ext;
   int flags;
} Address;

typedef struct addr_list {
   long magic;
   void *context;
   Address Addr;
   struct addr_list *pNext;
} AddressList;

typedef AddressList RecipientList;

typedef Address Recipient; /* Definition of Recipient type */

typedef Address Sender;

typedef struct {
   long magic;
   char *ControlFileName;
   char *BodyFileName;
   char *HeaderFileName;
   RecipientList *recipList; /* List of recipients */
   Sender *sender; /* Envelope sender */
   char *stage; /* Stage in processing */
   int flags;
   void *context; /* Internal context for */
                  /* operations, such as getNext routines */
} Message;
```

The `Message` structure represents the message itself. The other type used in the functions in this API is the `Recipient` structure, defined as a type of `Address`. For information about the magic value to use, see "Add and remove recipients."

To set or access this structure, use the functions associated with the `Message` type.

Warning This type definition is for informational purposes only. It does not reproduce the actual definition of the structure. §

Using Messaging Server Plug-in Functions

Messaging Server functions operate either on the `pblock` structure or the `Message` structure.

- One function operates on the parameter block, `pblock_findval`.
- These functions operate on the `Message` structure.

`AddRecipient`	`GetBodyFile`	`GetNextRecipient`
`DupMessage`	`GetFirstRecipient`	`GetRecipientAddress`
`FreeMessage`	`GetHeaderFile`	`RemoveRecipient`

This list summarizes the operations the Messaging Server API can perform. Each operation links to a further explanation that includes the function you need to perform it.

- Find a Messaging Server entry
- Duplicate the message
- Free the message
- Get message recipients
- Add and remove recipients
- Get the recipient's address
- Duplicate the message

- Get header and body files

These examples demonstrate the operations above in more detail.

Find a Messaging Server entry

The `pblock_findval` function searches the hash table (`pblock` parameter) for the entry with the given name and returns its value or `null`. It is the only function that operates on the parameter block.

```
char *pblock_findval(char *name, pblock *pb);
```

Duplicate the message

To duplicate an existing `Message` structure, use `DupMessage`.

```
Message *DupMessage (Message *pMessage);
```

This function creates a new instance of the message in the Mail Server. Mail Server automatically allocates header and body files for the new message and copies the contents of the header and body files of the original message. The function returns `null` if memory failures occur or if the message is not valid.

Free the message

The `FreeMessage` function frees the resources associated with the message.

```
void FreeMessage (Message *pMessage);
```

Get message recipients

To access the recipient list of the input message, use the `GetFirstRecipient` and `GetNextRecipient` functions.

`GetFirstRecipient` gets a pointer to the first recipient in the message.

```
Recipient *GetFirstRecipient (Message *pMessage);
```

`GetNextRecipient` gets the second, third, and all subsequent recipients.

```
Recipient *GetNextRecipient (Message *pMessage);
```

Add and remove recipients

To add or delete recipients on the recipient list, use the `AddRecipient` or `RemoveRecipient` function.

```
int AddRecipient (Message *pMessage,
                  Recipient *pRecipient);
```

```
int RemoveRecipient (Message *pMessage,
                     Recipient *pRecipient);
```

`AddRecipient` succeeds if both input parameters are valid and no memory errors occur. `RemoveRecipient` always succeeds.

This example shows how you can use `AddRecipient`. First, define the new recipient with any required flags or extension.

```
new_rcpt.Addr821 = (char*)malloc (strlen ("<testuser@test.com>") + 1);
strcpy(new_rcpt.Addr821, "<testuser@test.com>");
new_rcpt.Ext = NULL;// Extension is null
new_rcpt.flags = 0; // No flags used in this example
```

Set this required magic value.

```
new_rcpt.magic = 0xdeadbeef;//Use only this magic value
```

Make the `AddRecipient` call. Add the `free` command to free the recipient when it is finished.

```
AddRecipient(ppInMsg[0],&new_rcpt); //Add the new recipient
free (new_rcpt.Addr821);
```

To see this sequence as part of a sample plug-in DLL, see "Messaging Server Sample Plug-in."

Get the recipient's address

To get the RFC 821 address of the recipient, for example, `foo@somewhere.org`, use `GetRecipientAddress`.

```
char *GetRecipientAddress (Recipient *pRecipient);
```

Get header and body files

To find, open, and rewrite or encode the header and body files of the message, use the `GetHeaderFile` and `GetBodyFile` functions.

```
char *GetHeaderFile (Message *pMessage);
char *GetBodyFile (Message *pMessage);
```

These functions return the full path name to the files that contain the RFC 822 header and RFC 822 body portions of the message, respectively. `GetHeaderFile` and `GetBodyFile` always succeed if the message is valid.

Generating the Shared Object

The shared object is the dynamically loadable code that implements the additional plug-in functionality. It is passed in the code that specifies plug-in functionality in the plug-in configuration file as the `SharedObject` parameter. This code is described in "Plug-in Entry Points."

When a shared object is configured for a stage or entry point, either `PostSmtpAccept` or `PreSmtpDeliver`, every message that enters this stage goes through this shared object. For example, if a shared object is configured with the `PreSmtpDeliver` entry point, then all the outbound messages go through the entry point for that shared object.

The Messaging Server calls the shared object and waits for a return before calling it again. Messaging Server 3.0 is single-threaded. Plug-ins for 3.x servers can have globals, and should handle restartable system calls, such as signals.

The following `ld` lines are examples of generating the shared object. In these examples, the source code file implementing the plug-in functionality is `plugin.c`, and the object file is `plugin.o`.

Unix

`Solaris` For `gcc` version 2.6.3:

```
gcc -Wal -fpic -o plugin.o -c plugin.c
# ld -G -o plugin.so plugin.o
```

`HP-UX 9.05` Ensure that `libNSmail.sl` is present in the default search path of `ld`.
For native compiler:

```
cc +z -Aa -o plugin.o -c plugin.c
# ld -b -o plugin.sl -lNSmail plugin.o
```

Windows NT

```
cl -DWIN32 -D_WINDOWS plugin.c /link /dll /out:plugin.dll
```

```
NetscapeMTA30.lib
```

If `NetscapeMTA30.lib` is not in the current directory, you must specify the complete.

Chapter

3

Messaging Server Plug-in API Reference

Functions

AddRecipient	GetHeaderFile
DupMessage	GetNextRecipient
FreeMessage	GetRecipientAddress
GetBodyFile	pblock_findval
GetFirstRecipient	RemoveRecipient

Structures

Message	pblock	Recipient

Result Codes

This chapter is a reference to Messaging Server functions, structures, and result codes.

For Messaging Server program definitions, see the `plugin.h` header file. You can find this file at `http://help.netscape.com/products/server/messaging/3x/info/apiheader.html`.

Functions

The Messaging Server Plug-in API includes the following functions. This table lists the functions and the data structure on which each operates.

Function	Description	Structure
`AddRecipient`	Adds a recipient to the message.	`Message` `Recipient`
`DupMessage`	Duplicates the message.	`Message`
`FreeMessage`	Frees the resources associated with a message.	`Message`
`GetBodyFile`	Gets the body file associated with the message.	`Message`
`GetFirstRecipient`	Gets the first recipient in the message.	`Message`
`GetHeaderFile`	Gets the header file associated with the message.	`Message`
`GetNextRecipient`	Gets all subsequent recipients in the message.	`Message`
`GetRecipientAddress`	Gets the address of the specified recipient.	`Recipient`
`pblock_findval`	Finds the entry with the given name.	`pblock`
`RemoveRecipient`	Deletes recipients from the message.	`Message` `Recipient`

Messaging Server Plug-in API functions that return pointers to structures or to strings return `null` on error. Functions that return an integer return 0 on success and a negative value on error.

All input parameters are validity checked for memory corruption, and return an appropriate error if necessary.

AddRecipient

Adds the specified recipient to a message.

Syntax

```
#include <plugin.h>
int AddRecipient (Message *pMessage, Recipient *pRecipient);
```

Parameters The function has the following parameters:

`pMessage`	Pointer to the name of the `Message` structure.
`pRecipient`	Pointer to the address of the `Recipient` structure.

Returns

- If successful, the function returns 0.
- If unsuccessful, the function returns negative values.

Description This function adds recipients to the recipient list of the message structure. You also use this function to create a new recipient. `AddRecipient` succeeds if both input parameters are valid and no memory errors occur.

To add a recipient, you must define the magic value `0xdeadbeef`. For an example that shows how to define this value, see "Add and remove recipients." To see how `AddRecipient` is used in a sample plug-in DLL, see "Messaging Server Sample Plug-in."

This function operates on the `Message` and `Recipient` structures.

To delete recipients from the recipient list, use `RemoveRecipient`.

See Also `GetFirstRecipient`, `GetNextRecipient`, `GetRecipientAddress`, `RemoveRecipient`, `Recipient`, `Message`

DupMessage

Duplicates the specified message.

Syntax

```
#include <plugin.h>
Message *DupMessage (Message *pMessage);
```

Parameters The function has the following parameter:

`pMessage`	Pointer to the name of the `Message` structure.

Returns

- If successful, the function returns a new instance of the message in the Mail Server.
- If memory failures occur or if the message is not valid, the function returns `null`.

Description This function duplicates an existing `Message` structure. When this function returns, a separate instance of an identical message is created in the Mail Server. Mail Server automatically does the following:

- Allocates header and body files for the new message.
- Copies the contents of the header and body files of the message pointed to by the input `pMessage` argument.

If the pointer returned by `DupMessage` is returned in the `OutMessage` output parameter for the plug-in function, the Mail Server frees it. If the plug-in decides not to return the message it created after calling `DupMessage`, it should call `FreeMessage` for that pointer. For information about the plug-in function, see “Plug-in Function Format and Result Codes.”

For more information, see “Duplicate the message.”

This function operates on the `Message` structure.

See Also `FreeMessage`, `Message`

FreeMessage

Frees the resources associated with a message.

Syntax

```
#include <plugin.h>
void FreeMessage (Message *pMessage);
```

Parameters The function has the following parameter:

`pMessage`	Pointer to the `Message` structure with which the resources are associated.

Description This function frees the resources associated with the `Message` structure indicated in the `pMessage` parameter. It operates only on the in-memory-copy of a message, and does not touch anything on the disk.

If a plug-in decides not to return the message it created after calling `DupMessage`, it should call `FreeMessage` for that pointer. For information about the plug-in function, see "Plug-in Function Format and Result Codes."

For more information, see "Free the message."

This function operates on the `Message` structure.

See Also `DupMessage`, `Message`

GetBodyFile

Gets the name of the body file associated with the specified message.

Syntax

```
#include <plugin.h>
char *GetBodyFile (Message *pMessage);
```

Parameters The function has the following parameter:

`pMessage`	Pointer to the name of the `Message` structure.

Returns

- If successful, the function returns the filename string.
- If unsuccessful, the function returns `null`.

Description This function returns the name of the body file associated with the `Message` structure indicated by the `pMessage` parameter.

`GetBodyFile` returns a character string that contains the full path name to the file that contains the RFC 822 body portion of the message. The third-party API can use these names to open and rewrite or encode the content part of the message. This is always successful when the `Message` structure is valid.

For more information, see "Get header and body files." To see how this function is used in a sample plug-in DLL, see "Messaging Server Sample Plug-in."

This function operates on the `Message` structure.

See Also `GetHeaderFile`, `Message`

GetFirstRecipient

Gets a pointer to the first recipient in the message structure.

Syntax

```
#include <plugin.h>
Recipient *GetFirstRecipient (Message *pMessage);
```

Parameters The function has the following parameter:

`pMessage`	Pointer to the name of the `Message` structure.

Returns

- If successful, the function returns the address of the recipient.
- If there are no remaining recipients or if the message is empty or invalid, the function returns `null`.

Description This function returns a pointer to the first recipient in the message represented by the `pMessage` parameter. Any subsequent call to `GetFirstRecipient` resets the retrieval of recipients so that calls to `GetNextRecipient` return the second, third, and all subsequent recipients.

To traverse the recipient list of the input message, use the `GetFirstRecipient` and `GetNextRecipient` pair. The functions do not allocate any storage; both return `null` if the message is empty or malformed.

For more information, see "Get the recipient's address." To see how this function is used in a sample plug-in DLL, see "Messaging Server Sample Plug-in."

This function operates on the `Message` structure.

See Also `AddRecipient`, `GetNextRecipient`, `GetRecipientAddress`, `RemoveRecipient`, `Message`

GetHeaderFile

Gets the name of the header file associated with the specified message.

Syntax

```
#include <plugin.h>
char *GetHeaderFile (Message *pMessage);
```

Parameters The function has the following parameter:

`pMessage`	Pointer to the name of the `Message` structure.

Returns

- If successful, the function returns the message header.
- If there are no remaining recipients, the function returns `null`.

Description This function returns the name of the header file associated with the message structure in the `pMessage` parameter.

`GetHeaderFile` returns a character string that contains the full path name to the file that contains the RFC 822 header portion of the message. The third-party API can use these names to open and rewrite or encode the content part of the message. This is always successful when the `Message` structure is valid.

For more information, see "Get header and body files." To see how this function is used in a sample plug-in DLL, see "Messaging Server Sample Plug-in."

This function operates on the `Message` structure.

See Also `GetBodyFile`, `Message`

GetNextRecipient

Gets the second, third, and all subsequent recipients in the message.

Syntax

```
#include <plugin.h>
Recipient *GetNextRecipient (RecipientList *pRecipientList);
```

Parameters The function has the following parameter:

`pRecipientList`	Pointer to the list of recipients in the `Message` structure.

Returns

- If successful, the function returns the address of the recipient.
- If there are no remaining recipients or if the message is empty or malformed, the function returns `null`.

Description Call this function after `GetFirstRecipient` to return the second, third, and all subsequent recipients.

To traverse the recipient list of the input message, use the `GetFirstRecipient` and `GetNextRecipient` pair. These functions do not allocate any storage; both return `null` if the message is empty or malformed.

For more information, see "Add and remove recipients."

This function operates on the `Message` structure.

See Also `AddRecipient`, `GetFirstRecipient`, `GetRecipientAddress`, `RemoveRecipient`, `Recipient`

GetRecipientAddress

Gets the address of the specified recipient.

Syntax

```
#include <plugin.h>
char *GetRecipientAddress (Recipient *pRecipient);
```

Parameters The function has the following parameter:

`pRecipient`	Pointer to the address of the `Recipient` structure.

Returns

- If successful, the function returns the address o the recipient.
- If there are no remaining recipients, the function returns `null`.

Description This function returns the RFC 821 address string contained in the `Recipient` structure pointed to by `pRecipient`, for example, `<foo@somewhere.org>`.

For more information, see "Get the recipient's address." To see how this function is used in a sample plug-in DLL, see "Messaging Server Sample Plug-in." For the address definition, see RFC 821: "Simple Mail Transfer Protocol."

This function operates on the `Message` structure.

See Also `AddRecipient`, `GetFirstRecipient`, `GetNextRecipient`, `RemoveRecipient`, `Recipient`

pblock_findval

Finds the entry with the given name.

Syntax

```
#include <plugin.h>
char *pblock_findval(char *name, pblock *pb);
```

Parameters The function has the following parameters:

`name`	Name of the entry to find.
`pb`	Hash table used to search for the specified entry.

Returns

- If successful, the function returns the `value` portion of the entry with the given `name`.
- If unsuccessful, the function returns `null`.

Description This function uses the name value to find a `name-value` entry and return its `value` portion. It searches the parameter block specified in the `pb` parameter.

You can use this function to find the value of the `name=value` parameter in the plug-in entry point code described in "Plug-in Entry Points." This parameter represents name-value pairs that are passed to the function to add functionality.

The `pblock_findval` function operates on the `pblock`, or parameter block, structure. `pblock` is a hash table keyed on the name string, which maps name strings onto their value character strings.

For more information, see "Find a Messaging Server entry" and "Parameter Block."

See Also `pblock`

RemoveRecipient

Deletes recipients from the message.

Syntax

```
#include <plugin.h>
int RemoveRecipient (Message *pMessage, Recipient *pRecipient);
```

Parameters The function has the following parameters:

`pMessage`	Pointer to the name of the `Message` structure.
`pRecipient`	Pointer to the address of the `Recipient` structure.

Returns

- If successful, the function returns 0.
- If unsuccessful, the function returns negative values.

Description This function deletes recipients from the structure pointed to by the `pMessage` parameter.

The `AddRecipient` and `RemoveRecipient` pair is used to add and delete recipients in the recipient list. `RemoveRecipient` always succeeds. For more information, see "Add and remove recipients."

See Also `AddRecipient`, `GetFirstRecipient`, `GetNextRecipient`, `GetRecipientAddress`, `Recipient`, `Message`

Structures

This section lists the data structures used by the Messaging Server Plug-in API.

Structure	Description
`Message`	Represents a message.
`pblock`	Represents a hash table.
`Recipient`	Represents the address of the recipient of a message.

Warning The type definitions in this section are for informational purposes only. They do not reproduce the actual definitions of the structures. All types mentioned in this document are opaque, and members of the type defined structures are not exposed in the header files. The only access to the members is through the Messaging Server API. §

Message

Represents a message.

Syntax

```
#include <plugin.h>
struct message;
typedef struct Message Message;
```

Description This structure stores all attributes that define a single message. This includes all envelope information, such as envelope sender, envelope recipients, sender and per-recipient extensions, and so on, as well as RFC 822 attributes such as pointers to the RFC 822 headers and RFC 822 body.

To set or access these attributes, use the functions that operate on the `Message` structure, There are listed in the See Also for this reference entry.

For more information, see "Data Structures" and "Message Structure." For details about message definition, see RFC 822: "Standard for the Format of ARPA Internet Text Messages."

See Also `AddRecipient`, `DupMessage`, `FreeMessage`, `GetBodyFile`, `GetFirstRecipient`, `GetHeaderFile`, `GetNextRecipient`, `RemoveRecipient`

Recipient

Represents the address of the recipient of a message.

Syntax

```
#include <plugin.h>
typedef struct Address;
typedef Address Recipient;
```

Description The Recipient structure is a type of Address that represents the address of the recipient of a message. An address can be an RFC 821-compliant address or a newsgroup name string, for example: "devgroup@startup.com" or "alt.server.ideateam."

To set or access this structure, use the functions that operate on the Recipient structure.

For more information, see "Add and remove recipients" and "Get the recipient's address."

See Also AddRecipient, RemoveRecipient

pblock

Represents a hash table.

Syntax

```
#include <plugin.h>
struct pblock;
```

Description The parameter block represents a hash table that is keyed on the name string. The hash table, which contains hash table entry structures, maps the name strings to their value character strings.

The hash function is subject to change and is therefore not made known to application functions.

To access this structure, use the pblock_findval function.

For more information, see "Parameter Block."

See Also pblock_findval

Result Codes

The plug-in should return one of these codes to the Messaging Server.

Code	Description
`MSG_CONTINUE`	The plug-in succeeded. The Mail Server should continue to process the resulting messages. This code can occur at any stage of Mail Server processing.
`MSG_NOACTION`	The plug-in did not do anything. Continue to process the message.

Chapter
4

Messaging Server Sample Plug-in

This sample plug-in DLL shows how to add a recipient to a message that goes through the server.

The `testplugin` plug-in performs several operations:

- creates a log file and writes the first `Message` recipients into it
- adds a new recipient to the message
- adds text to the body and header of the message
- sends the message on its way

Note If you would like to see this plug-in in action, save and build the code in Plug-in Code. §

This list summarizes the plug-in operations. Each operation links to the line in the DLL that performs it.

- Include header files.
- Define the plug-in function.
- Define the recipients of the message and its body and temporary files.
- Create a log file.
- Find and print out the first recipient.

- Add a new recipient to the message.
- Add text to the body of the message.
- Add text to the header of the message.
- Send the message.

The testplugin DLL

Include header files

The plug-in DLL includes only the `plugin.h` and standard input/output header files.

```
#include <stdio.h>

#include "plugin.h"
```

Define the plug-in function

This function sets the hash table as `Config` and defines input (`ppInMsg`) and output (`pppOutMsg`) messages. The output parameter represents the result of the execution of the plug-in code.

The plug-in code should return either `MSG_CONTINUE` or `MSG_NOACTION` to the Messaging Server. `MSG_CONTINUE` indicates that the plug-in worked; message can go on to the Mail Server for further processing. `MSG_NOACTION` indicates that the plug-in did not do anything.

For information about defining this function, see “Plug-in Function Format and Result Codes.”

```
DLL int testplugin (pblock *Config, Message **ppInMsg, Message
***pppOutMsg)
```

Define the recipients of the message and its body and temporary files

```
{
LPSTR tmp;
LPSTR bodyfile;
Recipient *pRecip; //See the Recipient structure definition
Recipient new_rcpt;
FILE *body;
```

Create a log file

```
FILE *f = fopen ("c:\\log.out", "w");
```

Find and print out the first recipient

To get a pointer to the first recipient in the message, use `GetFirstRecipient`. The `fprintf` routine calls `GetRecipientAddress` to get the address for printing.

```
pRecip = GetFirstRecipient (ppInMsg [0]);
if (pRecip)
fprintf (f,"Recip #1 is %s\n", GetRecipientAddress(pRecip));
```

Add a new recipient to the message

Define the new recipient with any required flags or an extension.

```
new_rcpt.Addr821 = (char*)malloc (strlen ("<testuser@test.com>") + 1);
strcpy(new_rcpt.Addr821, "<testuser@test.com>");
new_rcpt.Ext = NULL;
new_rcpt.flags = 0;
```

This is the tricky part - you must set this magic value.

```
new_rcpt.magic = 0xdeadbeef;
```

Make the `AddRecipient` call.

```
AddRecipient(ppInMsg[0], &new_rcpt);

free (new_rcpt.Addr821);
```

For more information, see "Add and remove recipients."

Add text to the body of the message

Open the body and add some text. This sample either prints an error message if it can't add the text to the body file for some reason or adds, and prints, the lines of text.

```
bodyfile = GetBodyFile (*ppInMsg);
fprintf (f,"GetBodyFile = %s\n", bodyfile);
body = fopen (bodyfile,"a");

if (body == NULL)
{
 fprintf (f,"Couldn't append to body\n");
}

else
{
 fprintf (body,"These are some body new lines\n");
 fprintf (body,"These are some body new lines\n");
 fprintf (body,"These are some body new lines\n");
 fprintf (body,"These are some body new lines\n");
 fprintf (body,"These are some body new lines\n");
 fprintf (body,"These are some body new lines\n");
 fprintf (body,"These are some body new lines\n");
 fprintf (body,"These are some body new lines\n");
 fclose (body);
}
```

Add text to the header of the message

Next, open the header file and add some text. This works like the code for adding text to the body file. It either prints an error message if it can't add the text to the header file or adds, and prints, the lines of text.

```
tmp = GetHeaderFile (*ppInMsg);
fprintf (f, "GetHeaderFile = %s\n", tmp);
body = fopen (tmp,"a");

if (body == NULL)
{
 fprintf (f, "Couldn't append to header\n");
}

else
{
 fprintf (body,"X-NewHeader: this is my new header\n");
 fprintf (body,"X-NewHeader: this is my new header\n");
 fprintf (body,"X-NewHeader: this is my new header\n");
 fprintf (body,"X-NewHeader: this is my new header\n");
 fclose (body);
}
fclose(f);
```

Send the message

If the plug-in returns MSG_CONTINUE, the message continues on its way. For other return messages, see "Result Codes."

```
return MSG_CONTINUE;
```

Note You can find the code for this plug-in in Plug-in Code. §

Plug-in Code

```
/* If you want to see this plug-in in action, you can save,
*  build and run the plug-in code in this file. */
/*********************************************************************
* MESSAGING SERVER 3.5 - Sample Plugin DLL
*
* Platform: Windows NT
* Author: Sean Maurik
* Purpose: This plugin creates a log file and writes the first
*           Message recipients into it.  It then adds a new
*           recipient to the message and adds some text to the
*           body and header of the message before sending it on its way
*********************************************************************/
#include <stdio.h>
#include "plugin.h"
DLL int testplugin (pblock *Config,Message **ppInMsg,Message
***pppOutMsg)
{
 LPSTR tmp;
 LPSTR bodyfile;
 Recipient *pRecip;
 Recipient new_rcpt;
 FILE *body;
 /* Create a log file */
 FILE *f = fopen ("c:\\log.out","w");
 /* print out the first recipient */
 pRecip = GetFirstRecipient (ppInMsg [0]);
 if (pRecip)
  fprintf (f,"Recip #1 is %s\n",GetRecipientAddress(pRecip));
 /* now add a new recipient */
 new_rcpt.Addr821 = (char*)malloc (strlen ("<testuser@test.com>") + 1);
```

```
strcpy(new_rcpt.Addr821, "<testuser@test.com>");
new_rcpt.Ext     = NULL;
new_rcpt.flags   = 0;
/* this is the tricky part - you must have this magic value set */
new_rcpt.magic = 0xdeadbeef;
/* finally make the call */
AddRecipient(ppInMsg[0],&new_rcpt);
free (new_rcpt.Addr821);
/* Now open the body and add some text */
bodyfile = GetBodyFile (*ppInMsg);
fprintf (f,"GetBodyFile = %s\n",bodyfile);
body = fopen (bodyfile,"a");
if (body == NULL)
{
 fprintf (f,"Couldn't append to body\n");
}
else
{
 fprintf (body,"These are some body new lines\n");
 fprintf (body,"These are some body new lines\n");
 fprintf (body,"These are some body new lines\n");
 fprintf (body,"These are some body new lines\n");
 fprintf (body,"These are some body new lines\n");
 fprintf (body,"These are some body new lines\n");
 fprintf (body,"These are some body new lines\n");
 fprintf (body,"These are some body new lines\n");
 fclose (body);
}
/* open the header file and add some text*/
tmp = GetHeaderFile (*ppInMsg);
fprintf (f,"GetHeaderFile = %s\n",tmp);
body = fopen (tmp,"a");
```

```
 if (body == NULL)
 {
  fprintf (f,"Couldn't append to header\n");
 }
 else
 {
  fprintf (body,"X-NewHeader: this is my new header\n");
  fprintf (body,"X-NewHeader: this is my new header\n");
  fprintf (body,"X-NewHeader: this is my new header\n");
  fprintf (body,"X-NewHeader: this is my new header\n");
  fclose (body);
 }
 fclose(f);
 /* now return and the message will continue */
 return MSG_CONTINUE;
}
```

Index