Sendmail 8.13.3 Programmer's Guide

HP-UX 11i v1 and HP-UX 11i v2

Manufacturing Part Number: 5991-0705
February 2005

United States

© Copyright 2005 Hewlett-Packard Development Company L.P.

Legal Notices

The information in this document is subject to change without notice.

Hewlett-Packard makes no warranty of any kind with regard to this manual, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Hewlett-Packard shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Warranty

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

U.S. Government License

Proprietary computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Copyright Notice

Copyright © 2005 Hewlett-Packard Development Company L.P.
Reproduction, adaptation, or translation of this document without prior written permission is prohibited, except as allowed under the copyright laws.

Trademark Notices

UNIX® is a registered trademark in the United States and other countries, licensed exclusively through The Open Group.

Contents

Contents

Contents

Contents

Contents

Contents

About This Document

This document describes how to use Milter APIs with Sendmail 8.13.3 on your HP-UX 11i v1 and HP-UX 11i v2 operating systems.

It is assumed that the HP-UX 11i v1 or the HP-UX 11i v2 operating system software and the appropriate files, scripts, and subsets are installed on your system.

Intended Audience

This manual is intended for application developers who are responsible for developing filter applications using the Milter APIs. Developers are expected to have knowledge of operating system concepts, library functions, and C coding. They should also have knowledge of Transmission Control Protocol/Internet Protocol (TCP/IP) networking concepts, network configuration, and Sendmail basics. This manual is not a C, Sendmail, or TCP/IP tutorial.

What Is in This Document

Sendmail 8.13.3 Programmer's Guide is divided into several chapters, each of which contains information about Milter APIs.

Table 1 briefly describes each chapter.

Table 1 Document Contents

Chapter	Description
Introduction	Presents an overview of the Milter functionality and lists the components that the Sendmail 8.13.3 software contains.
Control Flow of Milter APIs	Describes the call order sequence of different Milter APIs. It also discusses multithreading, resource handling, and signal handling in Milter.
Configuring and Compiling Milter APIs	Discusses how to configure and compile Sendmail with a Milter application.

Table 1 **Document Contents (Continued)**

Chapter	Description
Milter APIs	Describes all Milter APIs.
Sample Program	Includes a sample filter program.

Related Documents

For more information on Sendmail 8.13.3, see the following documents:

- HP-UX Mailing Services Administrator's Guide at `http://www.docs.hp.com/hpux/netcom/index.html#Internet%20Services`.
- Request for Comments (RFC)

  Many sections of this manual refer to RFCs for more information about networking topics. These documents publicize Internet standards, new research concepts, and status memos about the Internet. You can access the full range of RFC documents and more information about the Internet Engineering Task Force (IETF) at the following URL:

  `http://www.ietf.org/rfc.html`

HP Encourages Your Comments

HP encourages any comments and suggestions you have on this document.

You can send your comments in the following ways:

- Internet electronic mail: netinfo_feedback@cup.hp.com
- A feedback form located at the following URL:

  http://docs.hp.com/assistance/feedback.html

Please include the following information when sending your feedback to us:

- The full title of the manual and the part number. (The part number appears on the title page of printed and PDF versions of a manual.)
- The section numbers and page numbers of the information on which you are commenting.

- The version of HP-UX you are using.

Please note that the HP-UX networking communications publications group does not provide technical support for HP products. If your inquiry concerns technical support for an HP product, please use the Assistance directory page located at: http://www.hp.com/ghp/assist/directory.html or call HP support at (208) 323-2551.

Typographic Conventions

This document uses the following typographic conventions:

$	A dollar sign represents the system prompt for the C and POSIX shells.
#	A number sign represents the superuser prompt.
file	Italic (slanted) type indicates document and book names.
`daemon`	Courier font type indicates daemons, files, commands, manual reference pages, and option names.
{|}	In syntax definitions, brackets indicate items that are optional and braces indicate items that are required.
(`Ctrl+A`)	This symbol indicates that you hold down the first named key while pressing the key or mouse button that follows the plus.
`Parameter`	Italic courier font type indicates input parameters for a function or API.
`Return Value`	Courier font type indicates values that a function returns.

1 Introduction

This chapter provides an overview of the Milter functionality introduced in Sendmail 8.13.3. In addition, the chapter briefly describes of the Milter architecture.

This chapter discusses the following topics:

NOTE All occurrences of Sendmail in this document refer to Sendmail 8.13.3.

Milter Overview

Sendmail 8.13.3 contains an advanced and effective mail filtering facility called Milter, which stands for Mail Filter. Milter is both a protocol and a library. Milter APIs provide an interface for third-party software to validate and modify messages as they pass through the mail transport system. Milter APIs enable filters to “listen in” to the SMTP conversation and modify Simple Mail Transfer Protocol (SMTP) responses.

To modify aspects of the message, you can call the Milter library functions that send special messages to Sendmail. The Milter library is multi-threaded and a given Sendmail installation can have multiple mail filters. Sendmail is single threaded but it forks into multiple processes. Sendmail uses mail filters to filter incoming SMTP messages.

Milter APIs provide the following benefits:

- Safety and security – You do not need root user privileges to run the filter processes. This feature simplifies coding and limits the impact of security flaws in the filter program.
- Reliability – Any failure in the Milter program does not affect Sendmail. When the Milter program fails, Sendmail either considers that the Milter program does not exist or considers that the required resource is unavailable.
- Simplicity – You can use Milter APIs to easily implement filters in Sendmail. To make the implementation easy, you can use threads by defining thread-clean interfaces that include local data hooks.
- Performance – A simple Milter program does not degrade the performance of Sendmail.

Following lists the types of Milter APIs:

- Library control functions
- Data access functions
- Message modification functions
- Other message handling functions
- Callbacks

Milter APIs operate in the following phases:

- At various stages of the SMTP conversation, Sendmail sends a message over the socket to the Milter program.
- The Milter library invokes a callback into your code and sends a reply message to Sendmail containing the return value from your callback.

Milter-Related Files

The Sendmail 8.13.3 depot contains C header files, libraries, and example programs. You can download the Sendmail 8.13 software depot, supported on HP-UX 11i v1 and HP-UX 11i v2, from the following URL:

`http://www.software.hp.com`

When you install the depot on your system, the depot installs the milter-related libraries and header files on your system. The milter-related library and header files supplied with Sendmail 8.13.3 are detailed in the following sections:

- "Header Files" on page 5
- "Library" on page 5

Header Files

Table 1-1 lists the milter-related header files included in the Sendmail 8.13.3 depot.

Table 1-1 Milter-Related Header Files

Header File	Description
`/usr/include/libmilter/mtapi.h`	Contains global definitions for mail filter library and mail filters.
`/usr/include/libmilter/mfdef.h`	Contains global definitions for mail filter and Sendmail.

Library

Sendmail 8.13.3 includes the 32-bit Milter library, `libmilter.a`. You can use this library to build filter applications.

Implementing Filtering Policies

Milter enables a system administrator to combine third-party filters with Sendmail to implement a desired mail filtering policy. For example, if a system administrator wants to perform the following tasks:

- Scan incoming mail for viruses on different platforms.
- Eliminate unsolicited commercial mail message.
- Append a mandatory footer to selected incoming messages

The system administrator can configure the Mail Transport Agent (MTA) to filter messages first through a server-based anti-virus engine, then through a large scale antispam service, and finally append the desired footer if the message still meets the requisite criteria. System administrators can add or change any filter independently.

You cannot control the overall mail filtering environment but system administrators can control the mail filtering environment. Particularly, the system administrator must decide which filters are run, in what order they are run, and how they communicate with Sendmail. You can select these parameters, as well as the actions to be taken when a filter becomes unavailable during Sendmail configuration.

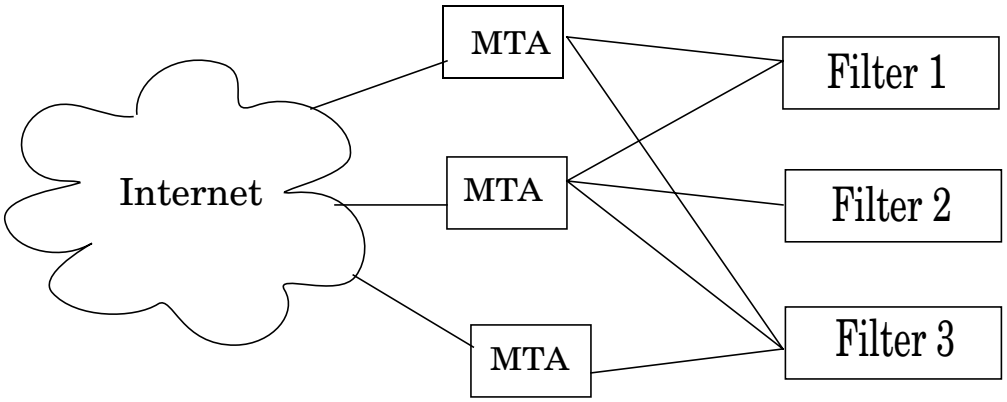
Communication Between an MTA and Milter

Filter applications run as separate processes outside the Sendmail address space. The benefits of filter applications running as separate processes are as follows:

- Filter applications do not need to run with "root" permissions, thereby, avoiding a large family of potential security problems.
- Failures in a particular filter do not affect Sendmail or other filters.
- Filter applications can have high performance because of the parallelism inherent in multiple processes.

Each filter can communicate with multiple MTAs at the same time over local or remote connections, using multiple threads of execution. Figure 1-1 illustrates a network of communication channels between the filters, its MTAs, and other MTAs on the network.

Figure 1-1 Communication Channel Between MTAs and Filters



The Milter library (`libmilter.a`) implements the communication protocol. It accepts connections from various MTAs and passes the relevant data to the filter through callbacks.

Before You Begin

Ensure that you have installed the Sendmail 8.13.3 depot from `http://www.software.hp.com` on your system. The Sendmail 8.13.3 depot contains the Milter archive library, `libmilter.a`, which you can use to build filter applications.

For more information on configuring and building filter applications, see “Configuring and Compiling Milter APIs” on page 55.

2 Milter APIs

This chapter discusses the different Milter APIs that Sendmail 8.13.3 includes.

It discusses the following topics:

Library Control APIs

This section describes the library control APIs that Sendmail 8.13.3 includes.

Filter applications use the library control APIs to provide the required information to Sendmail. Each of the library control APIs returns either `MI_SUCCESS` or `MI_FAILURE` to indicate the status of the operation. The library control APIs do not directly communicate with Sendmail, but they alter the state of the library.

Before handing control to `libmilter` (by calling `smfi_main()`), a filter application can call the following library control APIs to set the `libmilter` parameters:

- `smfi_register()`
- `smfi_opensocket()`
- `smfi_setconn()`
- `settimeout()`
- `smfi_setbacklog()`
- `smfi_setdbg()`
- `smfi_stop()`
- `smfi_main()`

The following sections discuss the library control APIs in detail.

The smfi_register() API

You can use the `smfi_register()` API to register a set of filter callbacks. You must call `smfi_register()` before calling the `smfi_main()` API. The `smfi_register()` API creates a filter using the information supplied by the `smfiDesc` argument. Do not call `smfi_register()` multiple times within a single process.

The declaration of `smfi_register()` is as follows:

```
#include <libmilter/mfapi.h>
int smfi_register(
        smfiDesc descr
);
```

Arguments

You must call `smfi_register()` with the following argument:

`descr` Specifies a filter descriptor of type `smfiDesc`, which describes the functions of the filter. The `smfiDesc` contains the following members:

```
struct smfiDesc
{
      char*xxfi_name;    /* filter name */
      intxxfi_version;/* version code -- do not change */
      unsigned longxxfi_flags;/* flags */

     /* connection info filter */
     sfsistat(*xxfi_connect)(SMFICTX *, char *, _SOCK_ADDR *);
     /* SMTP HELO command filter */
     sfsistat(*xxfi_helo)(SMFICTX *, char *);
     /* envelope sender filter */
     sfsistat(*xxfi_envfrom)(SMFICTX *, char **);
     /* envelope recipient filter */
     sfsistat(*xxfi_envrcpt)(SMFICTX *, char **);
     /* header filter */
     sfsistat(*xxfi_header)(SMFICTX *, char *, char *);
     /* end of header */
     sfsistat(*xxfi_eoh)(SMFICTX *);
     /* body block */
     sfsistat(*xxfi_body)(SMFICTX *, unsigned char *, size_t);
     /* end of message */
     sfsistat(*xxfi_eom)(SMFICTX *);
      /* message aborted */
      sfsistat(*xxfi_abort)(SMFICTX *);
      /* connection cleanup */
      sfsistat(*xxfi_close)(SMFICTX *);
};
```

A NULL value for any callback indicates that the filter does not want to process the given type of information and the callback returns only `SMFIS_CONTINUE`.

For more information on callbacks, see “Callbacks” on page 36.

Table 2-1 describes the bitwise OR of zero or more values, which the xxfi_flags field can contain. The table also describes the possible actions of the filter.

Table 2-1 The xxfi_flags Field Values

Flag	Description
SMFIF_ADDHDRS	Adds headers.
SMFIF_CHGHDRS	Changes and deletes headers.
SMFIF_CHGBODY	Replaces the body of the messageduring filtering. This can have significant performance impact if other filters filter the body after this filter.
SMFIF_ADDRCPT	Adds recipients to the message.
SMFIF_DELRCPT	Removes recipients from the message.

Return Values

The smfi_register() API returns MI_FAILURE because of the following reasons:

- Memory allocation failure
- Incompatible version or illegal value of flags

The smfi_setconn() API

You can use the smfi_setconn() API to set the socket through which a filter must communicate with Sendmail. You must call setconn() before calling the smfi_main() API.

The declaration of smfi_setconn() is as follows:

```
#include <libmilter/mfapi.h>
int smfi_setconn{
       char *oconn;
};
```

Do not run filters as a root user when communicating over UNIX® or local domain sockets.

If you use the Sendmail `RunAsUser` option, you must set the permissions for UNIX or local sockets to 0600 (read/write permission only for the owner of the socket) or 0660 (read/write permission for the owner and group of the socket). To determine the permission for a UNIX or local domain socket, you can use the `umask` command and set `umask` to 007 or 077.

Arguments

You must call `smfi_setconn()` with the following argument:

`oconn` Specifies the address of the desired communication socket. The address must be a NULL-terminated string in the following `proto:address` format:

- `{unix|local}:/path/to/file` – Specifies a named pipe.
- `inet:port@{hostname|ip-address}` – Specifies an IPV4 socket.
- `inet6:port@{hostname|ip-address}` – Specifies an IPV6 socket.

Return Value

The `smfi_setconn()` API does not fail on an invalid address. The failure is only detected in the `smfi_main()` API.

The smfi_settimeout() API

You can use the `smfi_settimeout()` API to set the connection timeout value of the filter. The connection timeout value specifies the number of seconds the `libmilter` library must wait for an MTA connection before timing out a socket. If the filter application does not call `smfi_settimout()`, the filter application uses a default timeout value of 7210 seconds.

The declaration of `smfi_settimout()` is as follows:

```
#include <libmilter/mfapi.h>
int smfi_settimeout(
         int otimeout
);
```

Arguments

You must call `smfi_settimeout()` with the following argument:

`otimeout`	Specifies the timeout value in seconds. You must specify a timeout value greater than 0 (zero). A value of 0 signifies that a filter does not to wait and, it does not signify that a filter must wait forever.

Return Value

The `smfi_settimeout()` API always returns `MI_SUCCESS` to the filter program.

The smfi_main() API

You can use the `smfi_main()` API to transfer control to the `libmilter` event loop. You must call `smfi_main()` after initializing a filter.

The declaration of `smfi_main()` is as follows:

```
#include <libmilter/mfapi.h>
int smfi_main(
);
```

The `smfi_main()` API does not contain any arguments.

Return Value

When `smfi_main()` fails to establish a connection, it returns `MI_FAILURE` to the filter application. The failure can occur because of many reasons, such as invalid address passed to `smfi_setconn()`. The reason for the failure is logged in the syslog file.

The `smfi_main()` API returns `MI_SUCCESS` on success.

The smfi_opensocket() API

You can use the `smfi_opensocket()` API to create the interface socket that MTAs use to connect to the filter.

You can call `smfi_opensocket()` only from the program mainline, before calling `smfi_main()`. You can use `smfi_opensocket()` to create the socket previously specified by a call to the `smfi_setconn()` API, which is the interface between MTAs and the filter. This allows the calling application to ensure that the socket can be created.

If you do not call `smfi_opensocket()`, `smfi_main()` will do so implicitly.

The declaration for `smfi_opensocket()` is as follows:

```
#include <libmilter/mfapi.h>
int smfi_opensocket(
bool rmsocket
);
```

Arguments

You must call `smfi_opensocket()` with the following argument

`smfi_opensocket`	Specifies the flag that indicates whether the library must try to remove any existing UNIX domain socket before trying to create a new one.

Return Value

`smfi_opensocket()` fails and returns `MI_FAILURE` because of the following reasons:

- The interface socket is not created.
- `rmsocket` is true and either the socket is not examined or exists, and is not removed.
- `smfi_setconn()` is not called.

`smfi_opensocket()` returns `MI_SUCCESS` on success.

The smfi_setdbg() API

You can use `smfi_setdbg()` to set the internal debugging or tracing level of the milter library to a new level to track the code details. You can use the `smfi_setdbg()` API to set the debugging or tracing level for the milter library. A level of 0 (zero) turns off debugging. If you increase the debugging level (more positive number), the details included in debugging also increases. A debugging value of 6 is the current, highest and useful debugging value.

The declaration of `smfi_setdbg()` is as follows:

```
#include <libmilter/mfapi.h>
int smfi_setdbg(
int level;
);
```

Argument

You must call `smfi_setdbg()` with the argument `level`, which specifies a new debugging level.

Return Value

By default, `smfi_setdbg()` returns `MI_SUCCESS` to the filter application.

The smfi_stop() API

You can use the `smfi_stop()` API to start an orderly shutdown of the Milter program. You can call `smfi_stop()` from any of the callbacks or any of the error-handling routines at any time. `smfi_stop()` causes each thread to finish its current connection and then exit the connection. When all the threads have exited, the call to the `smfi_main()` API returns to the calling program, which may then exit or warm restart the function (?). A filter application does not accept any new connection after calling `smfi_stop()`.

The declaration of `smfi_stop()` is as follows:

```
#include <libmilter/mfapi.h>
int smfi_stop(void);
```

Argument

You must call `smfi_stop()` with the argument `void`, which specifies that `smfi_stop()` does not accept any argument.

Return Values

`smfi_stop()` always returns `SMFI_CONTINUE` to the Milter program.

Following are additional points regarding `smfi_stop()`:

- Another internal routine may have already requested the Milter program to abort.
- Another routine may already have requested the Milter program to stop.
- You cannot cancel the stop process when it has begun.

Data Access APIs

You can call the data access APIs from within the filter-defined callbacks to access information about the current connection or message.

Following are the data access APIs:

- `smfi_getsymval()`
- `smfi_getpriv()`
- `smfi_setpriv()`
- `smfi_setreply()`
- `smfi_setmlreply()`

The following sections discuss the data access functions in detail.

The smfi_getsymval() API

You can use the `smfi_getsymval()` API to get the value of a Sendmail macro. The macros that are defined depend on when `smfi_getsymval()` is called. You can call `smfi_getsymval()` from any of the `xxfi_*` callbacks.

The declaration of `smfi_getsymval()` is as follows:

```
#include <libmilter/mfapi.h>
char* smfi_getsymval(
         SMFICTX *ctx,
         char *symname
);
```

Table 2-1 lists the Sendmail macros that you can use with the `xxfi_*` callbacks.

Table 2-2 Sendmail Macros

xxfi_* Callbacks	Sendmail Macros
`xxfi_connect()`	`daemon_name, if_name, if_addr, j, _`
`xxfi_helo()`	`tls_version, cipher, cipher_bits, cert_subject, cert_issuer`

Table 2-2 Sendmail Macros (Continued)

xxfi_* Callbacks	Sendmail Macros
xxfi_envfrom()	i, auth_type, auth_authen, auth_ssf, auth_author, mail_mailer, mail_host, mail_addr
xxfi_envrcpt()	rcpt_mailer, rcpt_host, rcpt_addr

All macros specified with the xxfi_connect() and xxfi_helo() callbacks are active from the point they are received until the end of the connection. All macros specified with the callback xxfi_envfrom() are active from the point they are received until the end of the message. All macros specified with the callbacks xxfi_envrcpt() are active for each recipient.

You can use the confMILTER_MACROS_* options in the Sendmail .mc file to change the macro list. Depending on when Sendmail sets the macros, you can determine the scope of these macros.

Arguments

You must call smfi_getsymval() with the following argument:

ctx — Specifies a opaque context structure.

symname — Denotes the name of a Sendmail macro. You can optionally enclose single letter macros and long macro names in braces ("{" and "}"), similar to the macros in the sendmail.cf file.

Return Value

smfi_getsymval() returns the value of the given macro as a null-terminated string or a NULL value if the macro is not defined.

The smfi_getpriv() API

You can use the smfi_getpriv() API to get the connection-specific data pointer for a connection. You can call smfi_getpriv() in any of the xxfi_* callbacks.

The declaration of smfi_getpriv() is as follows:

```
#include <libmilter/mfapi.h>
void* smfi_getpriv(
          SMFICTX *ctx
);
```

Argument

You must call `smfi_getpriv()` with the argument `ctx`, which specifies an opaque context structure.

Return Value

`smfi_getpriv()` returns the private data pointer stored by an earlier call to the `smfi_setpriv()` API, or NULL if none has been set.

The smfi_setpriv() API

You can use `smfi_setpriv()` to set the private data pointer for a connection. You can call `smfi_setpriv()` in any of the `xxfi_*` callbacks.

The declaration of `smfi_setpriv()` is as follows:

```
#include <libmilter/mfapi.h>
int smfi_setpriv(
       SMFICTX *ctx,
       void *privatedata
);
```

Only one private data pointer is available per connection; multiple calls to `smfi_setpriv()` with different values result in loss of previous values. Before a filter terminates, it must release the private data and set the pointer to NULL.

Arguments

You must call `smfi_setpriv()` with the following arguments:

`ctx`	Specifies the opaque context structure.
`privatedata`	Denotes a pointer to private data. This value is returned by subsequent calls to the `smfi_getpriv()` API using `ctx`.

Return Values

`smfi_setpriv()` returns `MI_FAILURE` if `ctx` is an invalid context structure. `smfi_setpriv()` returns `MI_SUCCESS` on success.

The smfi_setreply() API

You can use the `smfi_setreply()` API to set the default SMTP error reply code. Only `4xx` and `5xx` replies are accepted. You can call `smfi_setreply()` from any of the `xxfi_*` callbacks other than the `xxfi_connect()` callback. `smfi_setreply()` directly sets the SMTP error reply code for a connection. If subsequent error occurs because of an action taken by the filter, analyze the error code to identify the problem.

The declaration of `smfi_setreply()` is as follows:

```
#include <libmilter/mfapi.h>
int smfi_setreply(
         SMFICTX *ctx,
         char *rcode,
         char *xcode,
         char *message
);
```

Following are some points to consider regarding `smfi_setreply()`:

- Values passed to `smfi_setreply()` are not checked for standards compliance.
- The message parameter must contain only printable characters; other characters can result in undefined behavior. For example, CR or LR causes the call to fail, a single `%` (percentage) character causes the text to be ignored (if a `%` is required in a string, use `%%` similar to the usage in `printf (3)`).
- If the reply code (`rcode`) is `4XX` but `SMFI-REJECT` is used for the message, the custom reply is not used.

  Similarly, if the reply code (`rcode`) is `5XX` code but `SMFI_TEMPFAIL` is used for the message, the custom reply is not used.

NOTE

An error is not returned to the Milter program in neither of the previous two instances; `libmilter` silently ignores the reply code.

For details on reply codes and their meanings, see RFC 821 (*SIMPLE MAIL TRANSFER PROTOCOL*) or 2821 (*Simple Mail Transfer Protocol*) and RFC 1893 (*Enhanced Mail System Status Codes*) or 2034 (*SMTP Service Extension for Returning Enhanced Error Codes*).

- If the Milter program returns SMFI_TEMPFAIL and sets the reply code to 421, the SMTP server terminates the SMTP session with a 421 error code.

Arguments

You must call smfi_setreply() with the following arguments:

ctx	Specifies an opaque context structure.
rcode	Specifies a 3-digit (RFC 821 or RFC 2821) SMTP reply code as a null terminated string. You must not assign rcode to NULL, and rcode must be a valid 4XX or 5XX reply code.
xcode	Specifies an extended (RFC 1893 or RFC 2034) reply code. If xcode is NULL, the extended code is not used. xcode must conform to RFC 1893 or RFC 2034.
message	Specifies the text part of the SMTP reply. If the message is NULL, an empty message is used.

Return Value

smfi_setreply() fails because of the following reasons and returns MI_FAILURE:

- The rcode argument or xcode argument is invalid.
- A memory-allocation failure occurs.

smfi_setreply() returns MI_SUCCESS on success.

The smfi_setmlreply() API

You can use the smfi_setmlreply() API to set the default SMTP error reply code to a multi-line response. You can set only 4xx and 5xx reply codes.

You can call smfi_setmlreply() from any of the xxfi_* callbacks except the xxfi_connect() callback. smfi_setmlreply() directly sets the SMTP error reply code for a connection to the given lines after xcode. You must terminate the list of arguments that you pass to smfi_setmlreply() with a NULL value. The error code is used on subsequent error replies resulting from actions taken by the filter program.

The declaration of smfi_setmlreply() is as follows:

```
#include <libmilter/mfapi.h>
int smfi_setmlreply(
        SMFICTX *ctx,
        char *rcode,
        char *xcode,
        ...
);
```

Following are some points to consider regarding `smfi_setmlreply()`:

- Values passed to `smfi_setmlreply()` are not checked for standards compliance.
- The message parameter must contain only printable characters; other characters can result in a undefined behavior. For example, CR or LR causes the call to fail, a single `%` (percentage) character causes the text to be ignored (if a `%` is required in a string, use `%%` similar to the usage in `printf (3)`).
- If the reply code (`rcode`) is `4XX` but `SMFI-REJECT` is used for the message, the custom reply is not used.

  Similarly, if the reply code (`rcode`) is `5XX` code but `SMFI_TEMPFAIL` is used for the message, the custom reply is not used.

NOTE An error is not returned to the Milter program in neither of the previous two cases; `libmilter` silently ignores the reply code.

  For details about reply codes and their meanings, see RFC 821 or 2821, and RFC 1893 or RFC 2034.

- If the Milter program returns `SMFI_TEMPFAIL` and sets the reply code to `421`, the SMTP server terminates the SMTP session with a `421` error code.

Arguments

You must call `smfi_setmlreply()` with the following argument:

`ctx`	Specifies an opaque context structure.
`rcode`	Specifies the 3-digit SMTP reply code as specified in RFC 821 (*Simple Mail Transfer Protocol*) or 2821 (*Simple Mail Transfer Protocol*). `rcode` is a null-terminated string and must be a valid `4XX` or `5XX` reply code. You must not set `rcode` to a NULL value.

`xcode`	Specifies the extended reply code as specified in RFC 1893 (*Enhanced Mail System Status Codes*) or 2034 (*SMTP Service Extension for Returning Enhanced Error Codes*). If `xcode` is NULL, an extended code is not used; otherwise, `xcode` must conform to RFC 1893 or RFC 2034.
...	Specifies the remaining arguments, that are single lines of text (upto 32 arguments), which is used as the text part of the SMTP reply. The list must be NULL terminated.

Return Values

`smfi_setmlreply()` fails because of the following reasons and returns `MI_FAILURE`:

- The `rcode` or `xcode` argument is invalid.
- A memory-allocation failure occurs.
- A text line contains a carriage return (CR) or line feed (LF).
- The length of any text is more than the `MAXREPLYLEN` (980) value.
- The reply contains more than 32 lines of text.

`smfi_setmlreply()` returns `MI_SUCCESS` on success.

Message Modification APIs

The message modification APIs change the contents and attributes of a message. These APIs include additional communication with the MTA and return either `MI_SUCCESS` or `MI_FAILURE` to indicate the status of the operation. You can call these APIs only in the `xxfi_eom()` callback.

A filter program must set the appropriate flag in the description passed to the `smfi_register()` API to call any message modification function. The MTA treats a call to the function as a failure of the filter program and terminates its connection when a filter program does not set the appropriate flag.

The status returned indicates only whether the message of the filter was successfully sent to the MTA and does not indicate whether the MTA has performed the requested operation. For example, when the `smfi_addheader()` API is called with an illegal header name, `smfi_addheader()` returns `MI_SUCCESS` even though the MTA can later refuse to add the illegal header.

Following are the message modification APIs:

- `smfi_addheader()`
- `smfi_chgheader()`
- `smfi_insheader()`
- `smfi_addrcpt()`
- `smfi_delrcpt()`
- `smfi_replacebody()`
- Other message modification APIs

The following sections discuss the message modification APIs in detail.

The smfi_addheader() API

You can use the `smfi_addheader()` API to add a header to the current message. You can call `smfi_addheader()` only from the `xxfi_eom()` callback.

The declaration for `smfi_addheader()` is as follows:

```
#include <libmilter/mfapi.h>
```

```
int smfi_addheader(
          SMFICTX *ctx,
          char *headerf,
          char *headerv
);
```

Following are some points to consider regarding `smfi_addheader()`:

- `smfi_addheader()` does not change existing headers of a message. To change the current value of a header, use `smfi_chgheader()`.
- A filter which calls `smfi_addheader()` must set the `SMFIF_ADDHDRS` flag in the `smfiDesc_str` passed to the `smfi_register()` API.
- For `smfi_addheader()`, the order of the filter program is important. Later filters will observe the header changes made by earlier filters.
- The filter program does not check the name and the value of the header for standards compliance. However, each line of the header must be less than 2048 characters. If you require longer headers, use multiline headers. To make a multiline header, insert a LF (ASCII `0x0` character or `\n` in C language) followed by at least a white space character, such as a space (ASCII `9x20`) or a tab (ASCII `0x09` or `\t` in C language) character.

  You must not precede the LF with a CR (ASCII `0x0d` character) because the MTA adds the CR automatically. You must ensure that you do not violate any standards.

Arguments

You must call `smfi_addheader()` with the following arguments:

`ctx`	Specifies an opaque context structure.
`headerf`	Specifies the header name, which is a non-NULL string.
`headerv`	Specifies the header value. `headerv` is a non-NULL, null-terminated string. `headerv` can also be an empty string.

Return Values

`smfi_addheader()` fails because of the following reasons and returns `MI_FAILURE`:

- `headerf` or `headerv` value is NULL.
- Adding headers in the current connection state is invalid.
- Memory allocation fails.
- Network error occurs.
- `SMFIF_ADDHDRS` is not set when the `smfi_register()` API is called.

`smfi_addheader()` returns `MI_SUCCESS` on success.

Example

Following is an example for `smfi_addheader()`:

```
int ret;
    SMFICTX *ctx;

...

ret = smfi_addheader(ctx, "Content-Type",
    "multipart/mixed;\n\tboundary='foobar'");
```

The smfi_chgheader() API

You can use the `smfi_chgheader()` API to change or delete a message header. You can call `smfi_chgheader()` only from the `xxfi_eom()` callback.

The declaration of `smfi_chgheader()` is as follows:

```
#include <libmilter/mfapi.h>
int smfi_chgheader(
         SMFICTX *ctx,
         char *headerf,
         mi_int32 hdridx,
         char *headerv
);
```

Following are some points to consider regarding `smfi_chgheader()`:

- While you can use `smfi_chgheader()` to add new headers, it is efficient to use the `smfi_addheader()` API.
- A filter program that calls the `smfi_chgheader()` API must set the `SMFIF_CHGHDRS` flag in the `smfiDesc_str` passed to the `smfi_register()` API.

- The filter order is important for the `smfi_chgheader()` API. A filter application placed later in the sequence observes the changes already done by earlier filters.
- The filter program does not check the name and the value of the header for standards compliance. However, each line of the header must be less than 2048 characters. If you require longer headers, use multiline headers. To make a multiline header, insert a LF (ASCII `0x0` character or `\n` in C language) followed by at least a white space character, such as a space (ASCII `9x20`) or a tab (ASCII `0x09` or `\t` in C language) character.

  You must precede the LF with a CR (ASCII `0x0d` character) because the MTA adds the CR automatically. You must ensure that you do not violate any standards.

Arguments

You must call `smfi_chgheader()` with the following arguments:

`ctx`	Specifies an opaque context structure.
`headerf`	Specifies the header name, which is a a non-NULL, null-terminated string.
`hdridx`	Specifies the header index value (1-based). A `hdridx` value of 1 modifies the first occurrence of a header named `headerf`. If `hdridx` is greater than the number of occurrences of `headerf`, a new copy of `headerf` is added.
`headerv`	Specifies the new value of the given header. A value of NULL to `headerv` implies that you must delete the header.

Return Values

`smfi_chgheader()` fails because of the following reasons and returns `MI_FAILURE`:

- `headerf` is NULL.
- Modifying headers in the current connection state is invalid.
- Memory allocation failure.
- Network error occurs.
- `SMFIF_CHGHDRS` is not set when `smfi_register()` is called.

smfi_chgheader() returns MI_SUCCESS on success.

Example

Following is an example of smfi_chgheader():

```
int ret;
SMFICTX *ctx;

...

ret = smfi_chgheader(ctx, "Content-Type", 1,
                     "multipart/mixed;\n\tboundary='foobar'");
```

The smfi_insheader() API

You can use the smfi_insheader() API to prepend a header to the current message. You can call smfi_insheader() only from the xxfi_eom() callback.

The declaration of smfi_insheader() is as follows:

```
#include <libmilter/mfapi.h>
int smfi_insheader(
          SMFICTX *ctx,
           int hdridx,
            char *headerf,
             char *headerv
);
```

Following are some points to consider regarding smfi_insheader():

- smfi_insheader() does not change the existing headers of a message. To change the current value of a header, use the smfi_chgheader() API.
- A filter application that calls the smfi_insheader() API must set the SMFIF_ADDHDRS flag in smfiDesc_str passed to the smfi_register() API.
- For smfi_insheader(), the order in which you place filter applications is important. Filter applications placed later in the sequence observe changes already done by earlier filter applications. If the value of hdridx is larger than the number of headers in the message, the header is simply appended. The filter application does not check the name and the value of the header for standards compliance. However, each line of the header must be less than 2048

characters. If you need longer headers, use a multiline header. To make a multiline header, insert a LF (an ASCII `0x0a` character, or `\n` in C) followed by at least one white space character, such as, a space (an ASCII `0x20` character) or tab (an ASCII `0x09` character, or `\n` in C).

You must precede the LF with a CR (an ASCII `0x0d` character) because the MTA adds this automatically. You must ensure that you do not violate any standards.

Arguments

You must call `smfi_insheader()` with the following arguments:

`ctx`	Specifies an opaque context structure.
`hdridx`	Specifies the location in the internal header list where you must insert this header. If the value is set to 0, `hdridx` is the first header.
`headerf`	Specifies the header name, which is a non-NULL, null-terminated string.
`headerv`	Specifies the header value, which is a non-NULL, null-terminated string. You can set `headerv` to an empty argument.

Return Values

`smfi_insheader()` fails because of the following reasons and returns `MI_FAILURE`:

- The `headerf` value or `headerv` value is NULL.
- Adding headers in the current connection state is invalid.
- Memory allocation fails.
- Network error occurs.
- `SMFIF_ADDHDRS` is not set when the `smfi_register()` is called.

`smfi_insheader()` returns `MI_SUCCESS` on success.

Example

Following is an example of `smfi_insheader()`:

```
int ret;
SMFICTX *ctx;
```

```
...

ret = smfi_insheader(ctx, 0, "First", "See me?");
```

The smfi_addrcpt() API

You can use the `smfi_addrcpt()` API to add a recipient for the current message. You can call `smfi_addrcpt()` only from the `xxfi_eom()` callback.

The declaration for `smfi_addrcpt()` is as follows:

```
#include <libmilter/mfapi.h>
int smfi_addrcpt(
        SMFICTX *ctx,
        char *rcpt
);
```

A filter program that calls `smfi_addrcpt()` must set the `SMFIF_ADDRCPT` flag in the `smfiDesc_str` structure passed to `smfi_register()`.

Arguments

You must call `smfi_addrcpt()` with the following arguments:

`ctx`	Specifies an opaque context structure.
`rcpt`	Specifies the new address of the recipient.

Return Values

`smfi_addrcpt()` fails because of the following reasons and returns `MI_FAILURE`:

- The `rcpt` value is NULL.
- Adding recipients in the current connection state is invalid.
- Network error occurs.
- The `SMFIF_ADDRCPT` flag is not set when the `smfi_register()` routine is called.

The smfi_delrcpt() API

You can use the `smfi_delrctp()` API to delete a recipient from the envelope of the current message. You can call `smfi_delrcpt()` only from the `xxfi_eom()` callback.

The declaration for smfi_delrctp() is as follows:

```
#include <libmilter/mfapi.h>
int smfi_delrcpt(
        SMFICTX *ctx;
        char *rcpt;
);
```

The address is not deleted if an address and its expanded form do not match.

Arguments

You must call smfi_delrcpt() with the following arguments:

ctx	Specifies an opaque context structure.
rcpt	Specifies the recipient address to be removed. The recipient address is a non-NULL, null-terminated string.

Return Values

smfi_delrcpt() fails because of the following reasons and returns MI_FAILURE:

- The rcpt variable is NULL.
- Deleting recipients in the current connection state is invalid.
- Invalid error occurs.
- The SMFIF_DELRCPT is not set when the smfi_register() routine is called.

smfi_delrcpt() returns MI_SUCCESS on success.

The smfi_replacebody() API

You can use the smfi_replacebody() API to replace the data in the message body. Use smfi_replacebody() only from the xxfi_eom() callback.

You must not call smfi_replacebody() more than once. If you call smfi_replacebody() more than once, the subsequent smfi_replacebody() calls append data to the new body of the message.

The declaration of smfi_replacebody() is as follows:

```
#include <libmilter/mfapi.h>
int smfi_replacebody(
        SMFICTX *ctx,
        unsigned char *bodyp,
        int bodylen
);
```

Following are some points to consider regarding `smfi_replacebody()`:

- As the message body can be very large, setting `SMFIF_CHGBODY` can significantly affect the performance of the filter program.
- If a filter program sets `SMFIF_CHGBODY` but does not call `smfi_replacebody()`, the original body remains unchanged.
- The filter order is important for `smfi_replacebody()`. Filters placed later in the sequence observe the changes created by earlier filters.

Arguments

You can call `smfi_replacebody()` with the following arguments:

`ctx`	Specifies an opaque context structure.
`bodyp`	Denotes a pointer to the start of the new body data, which need not be null-terminated. If you set `bodyp` to NULL, the length of the body is considered to be 0 (zero). The body data must be in CR or LF form.
`bodylen`	Specifies the number of data bytes pointed by `bodyp`.

Return Values

`smfi_replacebody()` fails because of the following reasons and returns `MI_FAILURE`:

- The value of `bodyp` is equal to NULL and the value of `bodylen` is greater than 0.
- Changing the body in the current connection state is invalid.
- Network error occurs.
- The `SMFIF_CHGBODY` is not set when the `smfi_register()` routine is called.

Other Message Handling APIs

The following APIs provide special case handling instructions for the Milter API or the MTA, without altering the content or status of the message:

- `smfi_progress()`
- `smfi_quarantine()`

You can call these APIs only in the `xxfi_eom()` callback. These APIs can invoke additional communication with the MTA. They return either `MI_SUCCESS` or `MI_FAILURE` to indicate the status of the operation.

The status returned by these functions indicate whether the message of the filter was successfully sent to the MTA and does not indicate whether the MTA performed the requested operation.

The smfi_progress() API

You can use the `smfi_progress()` API to notify an MTA that an operation is still working on a message causing the MTA to restart its timeout values. You can call `smfi_progress()` from the `xxfi_eom()` callback.

The declaration of `smfi_progress()` is as follows:

```
#include <libmilter/mfapi.h>
int smfi_progress(
        SMFICTX *ctx;
);
```

Argument

You must call `smfi_progress()` with the `ctx` argument, which specifies an opaque context structure.

Return Values

`smfi_progress()` returns `MI_FAILURE` on failure if a network failure occurs. `smfi_progress()` returns `MI_SUCCESS` on success.

The smfi_quarantine() API

You can use the `smfi_quarantine()` API to quarantine the message using the specific reason. You can call `smfi_quarantine()` only from the `xxfi_eom()` callback.

The declaration of `smfi_quarantine()` is as follows:

```
#include <libmilter/mfapi.h>
int smfi_quarantine(
        SMFICTX *ctx;
        char *reason;
);
```

Arguments

You must call `smfi_quarantine()` with the following arguments:

`ctx`	Specifies an opaque context structure.
`reason`	Specifies the quarantine reason, which is a non-NULL and non-empty null-terminated string.

Return Values

`smfi_quarantine()` returns `MI_FAILURE` on failure because of the following reasons:

- The `reason` argument is NULL or empty.
- Network error occurs.
- The `SMFIF_QUARANTINE` value is not set when the `smfi_register()` routine is called.

`smfi_quarantine()` returns `MI_SUCCESS` on success.

Callbacks

A filter application must implement one or more of the following callbacks, which are registered through the `smfi_register()` API:

- `xxfi_connect()`
- `xxfi_helo()`
- `xxfi_envfrom()`
- `xxfi_envrcpt()`
- `xxfi_header()`
- `xxfi_eoh()`
- `xxfi_body()`
- `xxfi_eom()`
- `xxfi_abort()`
- `xxfi_close()`

NOTE You can replace the `xx` portion in the callback name with the name of your Milter program.

The following sections discuss these callbacks in detail.

The xxfi_connect() Callback

The `xxfi_connect()` callback returns the `SMFIS_CONTINUE` value to the calling filter application. `xxfi_connect()` is called once during the start of each SMTP connection.

The declaration of `xxfi_connect()` is as follows:

```
#include <libmilter/mfapi.h>
sfsistat (*xxfi_connect)(
            SMFICTX     *ctx,
            char        *hostname,
             _SOCK_ADDR *hostaddr);
```

If an earlier filter application rejects a connection in its xxfi_connect() callback, the current filter does not call xxfi_connect().

Arguments

You must call xxfi_connect() with the following arguments:

ctx	Specifies an opaque context structure.
hostname	Specifies the host name of the message sender, as determined by a reverse lookup on the host address. If the reverse lookup fails, hostname contains the IP address of the message sender enclosed in square brackets. For example, [a.b.c.d], where a.b.c.d denotes the IP address.
hostaddr	Specifies the host address, as determined by a getpeername() call on the SMTP socket. The value of hostaddr is NULL if the type is not supported in the current version or if the SMTP connection is made through stdin.

The xxfi_helo() Callback

The xxfi_helo() callback handles the HELO and EHLO commands. xxfi_helo() returns the SMFIS_CONTINUE value to the calling filter application. xxfi_helo() is called when the client sends a HELO or EHLO command. You can therefore call xxfi_helo() multiple times or you can also refrain from calling this callback.

The declaration of xxfi_helo() is as follows:

```
#include <libmilter/mfapi.h>
sfsistat (*xxfi_helo)(
         SMFICTX * ctx,
         char * helohost
);
```

Arguments

You must call xxfi_helo() with the following arguments:

ctx	Specifies an opaque context structure.
helohost	Specifies the value that is passed to HELO or EHLO, which must be the domain name of the sending host.

The xxfi_envfrom() Callback

The xxfi_envfrom() callback handles the envelope FROM command. xxfi_envfrom() returns the SMFIS_CONTINUE value to the calling filter application. xxfi_envfrom() is called once during the beginning of each message and before calling the xxfi_envrcpt() callback.

The declaration of xxfi_envfrom() is as follows:

```
#include <libmilter/mfapi.h>
sfsistat (*xxfi_envfrom)(
          SMFICTX * ctx,
          char **argv
);
```

Arguments

You must call xxfi_envfrom() with the following arguments:

ctx	Specifies an opaque context structure.
argv	Specifies null-terminated SMTP command arguments. argv[0] denotes the address of the sender. Later arguments, such as argv[1], argv[2], denote ESMTP arguments. For more information on ESTMP responses, see RFC 1869 (*SMTP Service Extensions*).

Return Values

xxfi_envfrom() returns the following values:

SMFIS_TEMPFAIL	Rejects the sender and message with a temporary error. The filter application does not call the xxfi_abort() callback to abort the message and you can specify a subsequent new message.
SMFIS_REJECT	Rejects the sender and message. The filter application does not call the xxfi_abort() callback to abort the message and you can specify a subsequent new message.
SMFIS_DISCARD	Accepts and silently discards a message. The filter application does not call the xxfi_abort() callback to abort the message.

SMFIF_ACCEPT	Accepts the message. The filter application does not call the xxfi_abort() callback to abort the message.

The xxfi_envrcpt() Callback

The xxfi_envrcpt() API handles the envelope RCTP command. xxfi_envrcpt() returns SMFIS_CONTINUE to the calling filter application. You can call xxfi_envrcpt() once for every recipient. If a message contains multiple recipients, you can call xxfi_envrcpt() multiple times, immediately after the xxfi_envfrom() callback.

The declaration of xxfi_envrcpt() is as follows:

```
#include <libmilter/mfapi.h>
sfsistat (*xxfi_envrcpt)(
          SMFICTX * ctx,
          char ** argv
);
```

Arguments

You must call xxfi_envrcpt() with the following arguments:

ctx	Specifies an opaque context structure.
argv	Specifies null-terminated SMTP command arguments. argv[0] denotes the address of the recipient. Later arguments, such as argv[1], argv[2], denote ESMTP arguments. For more information on ESTMP responses, see RFC 1869 (*SMTP Service Extensions*).

Return Values

xxfi_envrcpt() returns the following values:

SMFIS_TEMPFAIL	Fails temporarily for a recipient but the filter application processes further recipients because the filter application does not call the xxfi_abort() callback to abort the message.
SMFIS_REJECTS	Rejects a recipient but the filter application processes further recipients because the filter application does not call the xxfi_abort() callback to abort the message.

SMFIS_DISCARD	Accepts and discards the message. The filter application does not call the xxfi_abort() callback to abort the message.
SMFIS_ACCEPT	Accepts the recipient. The filter application does not call the xxfi_abort() callback to abort the message.

The xxfi_header() Callback

The xxfi_header() handles the message header and returns the SMFIS_CONTINUE value to the calling filter application. You can call xxfi_header() multiple times after calling the xxfi_envrcpt() callback and before calling the xxfi_eoh() callback, and once for each message header. Later filter applications can observe the header changes or additions made by earlier filter applications.

The declaration of xxfi_header() is as follows:

```
#include <libmilter/mfapi.h>
sfsistat (*xxfi_header)(
         SMFICTX * ctx,
         char * headerf,
         char * headerv
);
```

Arguments

You must call xxfi_header() with the following arguments:

ctx	Specifies an opaque context structure.
headerf	Specifies the header field name.
headerv	Specifies the header field value. The header content can include folded white space, that is, multiple lines followed by a white space. The filter application removes the trailing line terminator (CR or LF).

The xxfi_eoh() Callback

The xxfi_eoh() callback handles the end of message headers and returns SMFIS_CONTINUE to the calling filter application. You must call xxfi_eoh() only once after all the headers are sent and processed.

Argument

You must call xxfi_eoh() with the ctx argument, which specifies an opaque context structure.

The xxfi_body() Callback

The xxfi_body() callback handles a portion of message body and returns the SMFIS_CONTINUE value to the calling filter application. The filter application calls xxfi_body() multiple times after calling the xxfi_eoh() callback and before calling the xxfi_eom() callback.

The declaration of xxfi_body() is as follows:

```
#include <libmilter/mfapi.h>
sfsistat (*xxfi_body)(
          SMFICTX * ctx,
          unsigned char * bodyp,
          size_t len
);
```

Following are some points to consider regarding xxfi_body():

- The bodyp argument points to a sequence of bytes and it is not a C string that is a sequence of characters terminated by a null character (\0). You must not use the normal C string functions, such as strlen() to modify the block of data. The byte sequence in the block can also contain \0 characters. If you add a trailing \0 character, C string functions can still fail to work in the block.
- Because message bodies can be large, defining xxfi_body() significantly impacts the filter performance.
- The filter application represents end-of-lines as received from the SMTP transaction (normally as CR or LF).
- Later filter applications observe body changes made by earlier filter applications.
- You can send message bodies in multiple portions with one call to xxfi_body() per portion.

Arguments

You must call xxfi_body() with the following arguments:

ctx	Specifies an opaque context structure.

bodyp	Specifies a pointer to the beginning of a block of body data. bodyp is not valid outside a call to the xxfi_body() callback.
len	Specifies the amount of data pointed by bodyp.

The xxfi_eom() Callback

The xxfi_eom() callback denotes the end of a message and returns the SMFIS_CONTINUE value to the calling filter application. xxfi_eom() is called once after all calls to the xxfi_body() callback for a given message.

The declaration of xxfi_eom() is as follows:

```
#include <libmilter/mfapi.h>
sfsistat (*xxfi_eom)(
         SMFICTX * ctx
);
```

A filter application must make all its modifications to the message headers, body, and envelope in xxfi_eom() callback. These modifications are made through the smfi_* APIs.

Argument

You must call xxfi_eom() API with the ctx argument, which specifies an opaque context structure.

The xxfi_abort() Callback

The xxfi_abort() callback handles the messages that are aborted. xxfi_abort() returns the SMFIS_CONTINUE value to the calling filter application. You can call xxfi_abort() any time while processing the message, that is between a message-oriented API and the xxfi_eom() callback.

The declaration of xxfi_abort() is as follows:

```
#include <libmilter/mfapi.h>
sfsistat (*xxfi_abort)(
         SMFICTX * ctx
);
```

Following are some points to consider regarding xxfi_abort():

- xxfi_abort() must reclaim any resource allocated on a per-message basis and must be tolerant of being called between any two message-oriented callbacks.
- Calls to xxfi_abort() and xxfi_eom() are mutually exclusive.
- xxfi_abort() is not responsible for reclaiming connection-specific data because xxfi_close() is always called when a connection is closed.
- Because xxfi_abort() aborts the message, the filter application ignores the return value of xxfi_abort().
- xxfi_abort() is called only if the message is aborted outside the control of the filter application and if the filter application has not completed its message-oriented processing. For example, if a filter has already returned the values SMFIS_ACCEPT, SMFIS_REJECT, or SMFIS_DISCARD from a message-oriented routine, xxfi_abort() is not called even if the message is aborted later outside its control.

Argument

You must call xxfi_abort() with the ctx argument, which specifies an opaque context structure.

The xxfi_close() Callback

The xxfi_close() callback denotes that the current connection is closed. It returns the SMFIS_CONTINUE value to the calling filter application. The filter application always calls xxfi_close() once at the end of each connection.

The declaration of xxfi_close() is as follows:

```
#include <libmilter/mfapi.h>
sfsistat (*xxfi_close)(
          SMFICTX * ctx
);
```

Following are some points to consider regarding xxfi_close():

- You can call xxfi_close() in any order, that is, you can call xxfi_close() even before calling xxfi_connect(). After establishing a connection with the filter application, if Sendmail decides to discard the traffic of a connection, Sendmail does not pass

data to the filter application until the client closes down the connection. This is this time when `xxfi_close()` is called to close the connection.

- `xxfi_close()` is called on close even if the previous mail transaction was aborted.
- `xxfi_close()` is responsible for freeing any resource allocated on a per-connection basis.
- The filter application ignores the return value of `xxfi_close()` because after the connection is closed, the return value does not hold any importance.

Argument

You must call `xxfi_close()` with the `ctx` argument, which specifies an opaque context structure.

3 Control Flow of Milter APIs

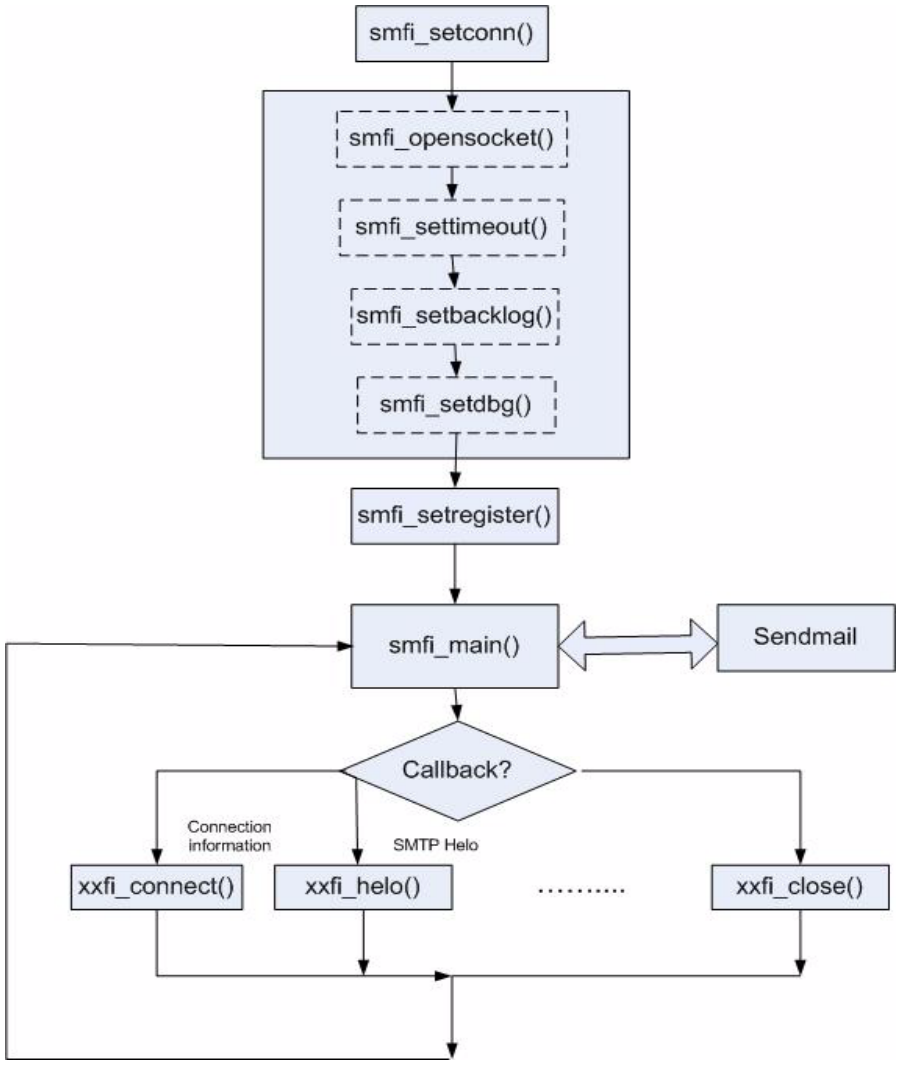
This chapter discusses the call order sequence and resource management techniques for Milter APIs. It also discusses the control flow, and performance enhancement of Milter APIs.

It discusses the following topics:

Call Order Sequence

Figure 3-1 illustrates the sequence in which the filter applications are called in a Milter program.

Figure 3-1 Milter Call Order Sequence



A filter application cannot process any message untill it registers its callbacks with Sendmail.

A filter application initiates a filter session using the `smfi_setconn()` API. The filter application initiates the session once and before calling the `smfi_main()` API.

The `smfi_setconn()` API sets the socket through which the filter application communicates with Sendmail. The filter application can optionally call the `smfi_opensocket()` API, which attempts to open the specified socket and ensures that the interface works properly. The filter application can also make optional calls to `smfi_settimeout()`, `smfi_setbacklog()`, and `smfi_setdbg()` before passing control to the `smfi_main()` API. After initiating the session and calling the optional APIs, the filter application must register with Sendmail, using the `smfi_register()` API, which informs Sendmail about the filter callbacks and the actual information the filter application requires. The filter application then passes control to the `smfi_main()` API.

The `smfi_main()` API starts the listener for the filter application and seeks for messages from Sendmail. The `smfi_main()` API makes respective calls to the callback functions before validating the message. For example, during a HELO message, `smfi_main()` invokes the filter callback `xxfi_helo()`.

Initialization Parameters for Filter Applications

In addition to initializing `libmilter`, a filter application must initialize the following parameters before calling the `smfi_main()` API:

- The callbacks the filter program requires to call and the types of message modification you intend to perform. You must initializing these parameters. For more information, see “The smfi_register() API” on page 11.
- The socket address to be used when communicating with an MTA. You must initialize these parameters. For more information, see “The smfi_setconn() API” on page 13.
- The number of seconds the filter application must wait for MTA connections before timing out. You can optionally initialize this parameter. For more information, see “The smfi_settimeout() API” on page 14.

A subsequent call to the `smfi_main()` API fails if the filter application fails to initialize `libmilter` and if one or more parameters passed by the filter application are invalid.

Sample Filter Pseudocode

The following pseudocode describes the filtering process from the perspective of a set of N MTAs, each corresponding to an SMTP connection.

```
For each of N connections
{
   For each filter
       process connection/helo (xxfi_connect, xxfi_helo)
MESSAGE:For each message in this connection (sequentially)
   {
       For each filter
                process sender (xxfi_envfrom)
       For each recipient
       {
                For each filter
                         process recipient (xxfi_envrcpt)
       }
       For each filter
       {
          For each header
                   process header (xxfi_header)
          process end of headers (xxfi_eoh)
          For each body block
                   process this body block (xxfi_body)
          process end of message (xxfi_eom)
        }
    }
    For each filter
             process end of connection (xxfi_close)
}
```

The callbacks within parenthesis are placed beside the processing stages in which they are called. If a callbacks is not defined for a particular stage, the filter application can bypass that stage. The filter application can abort processing at any time during a message, in which case the `xxfi_abort()` callback is invoked and the control returns to `MESSAGE`.

Sendmail contacts the filter applications in the order defined in the Sendmail configuration file.

To write a filter application, you must invoke different callbacks to process relevant parts of a message transaction. The Milter library then controls all sequencing, threading, and protocol exchange with Sendmail.

Table 3-1 outlines the control flow for a filter process and denotes different callbacks invoked during an SMTP transaction.

Table 3-1 Milter Callbacks Related to an SMTP Transaction

SMTP Commands	Milter Callbacks
(open SMTP connection)	`xxfi_connect()`
`HELO ...`	`xxfi_helo()`
`MAIL From: ...`	`xxfi_envfrom()`
`RCPT To: ...`	`xxfi_envrcpt()`
[more `RCPT`s]	[`xxfi_envrcpt()`]
`DATA`	
`Header: ...`	`xxfi_header()`
[more headers]	[`xxfi_header()`]
	`xxfi_eoh()`
`body...`	`xxfi_body()`
[more `body`...]	[`xxfi_body()`]
.	`xxfi_eom()`
`QUIT`	`xxfi_close()`
(close SMTP connection)	

Although Table 3-1 denotes only a single message, multiple messages can be sent in a single connection. The remote host or Sendmail can abort a message and connection anytime during the SMTP transaction. If the abort occurs during a message processing (that is, between the `MAIL` command and the final . command), the filter application calls the `xxfi_abort()` API. The filter application calls `xxfi_close()` any time when the connection closes.

Multithreading

A single filter process can handle any number of connections simultaneously. All the filtering callbacks must therefore be reentrant and they must use appropriate external synchronization methods to access global data. Because a one-to-one correspondence between the threads and connections (N connections mapped on to M threads, where M is less than or equal to N) does not exist, you must access connection-specific data through the handles provided by the Milter library. You must not rely on the thread-specific data blocks supplied by the library to store data blocks (for example, `pthread_getspecific()`) to store connection-specific data.

For more information on setting and getting connection-specific pointers, see “The smfi_setpriv() API” on page 20 and “The smfi_getpriv() API” on page 19, respectively.

Resource Management

You must deallocate per-connection resources because filter applications exist for a long time and they handle many connections. The lifetime of a connection depends on calls to the callbacks `xxfi_connect()` and `xxfi_close()`. For more information on message-oriented and connection-oriented APIs, see “Message Modification APIs” on page 25 and “Data Access APIs” on page 18, respectively. Only one connection-specific data pointer is available for each connection.

Each message is marked by calls to the `xxfi_envfrom()` and `xxfi_eom()` callbacks (or the `xxfi_abort()` callback), which implies that message-specific resources are allocated and reclaimed from these routines. Only one active message is available because the messages in a connection are processed sequentially by each filter, and it is associated with a given connection and filter (and connection-private data block). The filter application must access these resources through the `smfi_getpriv()` and `smfi_setpriv()` APIs and must reclaim the resources using the `xxfi_abort()` API.

Signal Handling

The Milter library, `libmilter.a`, manages signal handling, and the signals do not directly influence filter applications.

Sendmail 8.13.3 includes the following signal handlers:

- `Stop` – Specifies that new connections from the MTA are not accepted but existing connections are allowed to continue.
- `Abort` – Specifies that all filter applications will be stopped after the next communication with Sendmail happens.

4 Configuring and Compiling Milter APIs

This chapter discusses how to configure Milter APIs and to compile them with Sendmail.

This chapter discusses the following topics:

Compiling and Installing Your Filter

To compile a filter, you must complete the following steps:

1. Insert the `include` and Sendmail directories in your include path. For example, `-I/path/to/include -I path/to/sendmail`.
2. Ensure that the `libmilter.a` file is in your library path and link your filter application with this file. For example, you can use the `-lmilter` option to link your the filter application with this file.
3. Compile with `pthreads` either by using `-pthread` for `gcc` or by linking with a `pthreads` support library (`-lpthread`).

Following is an example of a command to compile a filter application:

```
# cc -I/path/to/include -I/path/to/sendmail -c myfile.c
```

where:

`myfile.c` specifies the name of the Milter program.

Following is an example of a command to link the filter application:

```
# cc -o myfilter [object-files] -L[library-location]
-lmilter -pthread
```

Configuring Milter in Sendmail

You must define a filter in your Sendmail configuration file and compile Sendmail.

To define a filter application in your Sendmail configuratin file, complete the following steps:

1. You must add filters to your `/usr/contrib/sendmail/etc/mail/cf/cf/generic-hpux-10.mc` file. You can use the following commands to configure filters in the `.mc` file:

   ```
   MAIL_FILTER ('name', 'equates')
   INPUT_MAIL_FILTER('name', 'equates')
   ```

   The `MAIL_FILTER()`command defines a filter with the given *name* and *equates*.

   For example, `MAIL_FILTER('archive', 'S=local:/var/run/archivesock, F=R')`

   where:

`S=local:/var/run/archivesock, F=R`	Specifies the equates.
`archive`	Specifies name of the filter application.

   This command creates the following equivalent entry in the `sendmail.cf` file:

   ```
   Xarchive, S=local:/var/run/archivesock, F=R
   ```

   The `INPUT_MAIL_FILTER()` command performs the same action as the `MAIL-FILTER` command but `INPUT_MAIL_FILTER` also populates the `m4` variable `confINPUT_MAIL_FILTERS` with the name of the filter such that the filter application is actually called by Sendmail.

2. You can define the `m4` variables or `cf` options to configure the Sendmail macros that are accessible through the `smfi_getsymval()` API.

Table 4-1 lists the different mf variables and cf options.

Table 4-1 The mf Variables and cf Options

The .mc File	The .cf File	Default Value
confMILTER_MACROS_CONNECT	Milter.macros.connect	j, _, {daemon_name}, {if_name}, {if_addr}
confMILTER_MACROS_HELO	Milter.macros.helo	{tls_version}, {cipher}, {cipher_bits}, {cert_subject}, {cert_issuer}
confMILTER_MACROS_ENVFROM	Milter.macros.envfrom	i, {auth_type}, {auth_authen}, {auth_ssf}, {auth_author}, {mail_mailer}, {mail_host}, {mail_addr}
confMILTER_MACROS_ENVRCPT	Milter.macros.envrcpt	{rcpt_mailer}, {rcpt_host}, {rcpt_addr}

Following are the equates that you can include in the .mc file:

- The required S= equate that specifies the socket where Sendmail must look for the filter.
- The optional F= equate that specifies flags.
- The optional T= equate that specifies timeouts.

All the equate names field names, and flag values are case sensitive.

Table 4-2 lists and describes the flag values for the F= equate.

Table 4-2 The F= Equate Values

Flag	Description
R	Rejects connection if the filter is not available.
T	Aborts connection temporarily if the filter is not available.

If a filter application is unavailable or unresponsive and you do not specify any flag in the /usr/contrib/sendmail/etc/mail/cf/cf/generic-hpux-10.mc file,

Sendmail 8.13.3 continues with the normal handling of the current connection. For every new connection, Sendmail 8.13.3 attempts to contact the filter application again.

Table 4-3 lists and describes the different fields in the T= equate.

Table 4-3 The T= Equate Values

Flag	Description
C	Specifies the timeout value for connecting to a filter application. If you set C to 0, the system connect() timeout value is used. The default timeout value for C is 5 minutes.
S	Specifies the timeout value for sending information from Sendmail to a filter application. The default value for S is 10 seconds.
R	Specifies the timeout value for reading reply from the filter application. The default value for R is 10 seconds.
E	Specifies the overall timeout value between sending the end-of-message to the filter and waiting for the final acknowledgment. The default value for E is 5 minutes.

A semicolon (;) separates each field because a comma (,) already separates the equates.

The separator between each field is a semicolon (;) because a comma (,) already separates the equates. The value of each field is a decimal number followed by a single letter designating the units (s for seconds and m for minutes).

Following is an example of a myconfig.mc file, which contains 3 filters, namely filter1, filter2, and filter3:

```
INPUT_MAIL_FILTER(‘filter1’, ‘S=unix:/var/run/f1.sock, F=R’)
INPUT_MAIL_FILTER(‘filter2’, ‘S=unix:/var/run/f2.sock, F=T, T=S:1s;R:1s;E:5m’)
INPUT_MAIL_FILTER(‘filter3’, ‘S=inet:999@localhost, T=C:2m’)

define(‘confINPUT_MAIL_FILTERS’, ‘filter2,filter1,filter3’)
```

Run the following command to generate the configuration file, `myconfig.cf`:

```
m4 ../m4/cf.m4 myconfig.mc > myconfig.cf
```

These macros add the following entries to your Sendmail configuration file (`sendmail.cf`):

```
Xfilter1, S=unix:/var/run/f1.sock, F=R
Xfilter2, S=unix:/var/run/f2.sock, F=T, T=S:1s;R:1s;E:5m
Xfilter3, S=inet:999@localhost, T=C:2m

O InputMailFilters=filter2,filter1,filter3
```

By default, the filters run in the order defined in the `.mc` file. However, because `confINPUT_MAIL_FILTERS` is defined, the filters are run in the order "`filter2, filter1, filter3`".

NOTE You can use the `MAIL_FILTER()` command, instead of the `INPUT_MAIL_FILTER()` command, to define a filter without adding it to the input filter list.

5 Sample Program

This chapter contains a sample C program for a filter application.

Milter Sample Program

Following is a sample filter program.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sysexits.h>
#include <unistd.h>

#include "libmilter/mfapi.h"

#ifndef bool
# define boolint
# define TRUE1
# define FALSE0
#endif /* ! bool */


struct mlfiPriv
{
        char*mlfi_fname;
        char*mlfi_connectfrom;
        char*mlfi_helofrom;
        FILE*mlfi_fp;
};

#define MLFIPRIV((struct mlfiPriv *) smfi_getpriv(ctx))

extern sfsistatmlfi_cleanup(SMFICTX *, bool);

/* recipients to add and reject (set with -a and -r options) */
char *add = NULL;
char *reject = NULL;

sfsistat
mlfi_connect(ctx, hostname, hostaddr)
         SMFICTX *ctx;
         char *hostname;
         _SOCK_ADDR *hostaddr;
{
```

```
        struct mlfiPriv *priv;
        char *ident;

        /* allocate some private memory */
        priv = malloc(sizeof *priv);
        if (priv == NULL)
        {
                /* can't accept this message right now */
                return SMFIS_TEMPFAIL;
        }
        memset(priv, '\0', sizeof *priv);

        /* save the private data */
        smfi_setpriv(ctx, priv);

        ident = smfi_getsymval(ctx, "_");
        if (ident == NULL)
                ident = "???";
       if ((priv->mlfi_connectfrom = strdup(ident)) == NULL)
       {
            (void) mlfi_cleanup(ctx, FALSE);
             return SMFIS_TEMPFAIL;
        }

        /* continue processing */
        return SMFIS_CONTINUE;
}

sfsistat
mlfi_helo(ctx, helohost)
          SMFICTX *ctx;
          char *helohost;
{
           size_t len;
           char *tls;
           char *buf;
           struct mlfiPriv *priv = MLFIPRIV;

           tls = smfi_getsymval(ctx, "{tls_version}");
           if (tls == NULL)
                   tls = "No TLS";
           if (helohost == NULL)
                   helohost = "???";
           len = strlen(tls) + strlen(helohost) + 3;
           if ((buf = (char*) malloc(len)) == NULL)
           {
```

```
                    (void) mlfi_cleanup(ctx, FALSE);
                     return SMFIS_TEMPFAIL;
                }
                snprintf(buf, len, "%s, %s", helohost, tls);
                if (priv->mlfi_helofrom != NULL)
                   free(priv->mlfi_helofrom);
                priv->mlfi_helofrom = buf;

                /* continue processing */
                return SMFIS_CONTINUE;
}

sfsistat
mlfi_envfrom(ctx, argv)
        SMFICTX *ctx;
        char **argv;
{
        int fd = -1;
        int argc = 0;
        struct mlfiPriv *priv = MLFIPRIV;
        char *mailaddr = smfi_getsymval(ctx, "{mail_addr}");

        /* open a file to store this message */
   if ((priv->mlfi_fname = strdup("/tmp/msg.XXXXXX")) == NULL)
    {
         (void) mlfi_cleanup(ctx, FALSE);
         return SMFIS_TEMPFAIL;
    }

    if ((fd = mkstemp(priv->mlfi_fname)) == -1)
    {
          (void) mlfi_cleanup(ctx, FALSE);
          return SMFIS_TEMPFAIL;
    }

    if ((priv->mlfi_fp = fdopen(fd, "w+")) == NULL)
    {
           (void) close(fd);
           (void) mlfi_cleanup(ctx, FALSE);
            return SMFIS_TEMPFAIL;
    }

     /* count the arguments */
     while (*argv++ != NULL)
             ++argc;
```

```
        /* log the connection information we stored earlier: */
        if (fprintf(priv->mlfi_fp, "Connect from %s (%s)\n\n",
            priv->mlfi_helofrom, priv->mlfi_connectfrom) == EOF)
        {
             (void) mlfi_cleanup(ctx, FALSE);
             return SMFIS_TEMPFAIL;
         }
         /* log the sender */
         if (fprintf(priv->mlfi_fp, "FROM %s (%d argument%s)\n",
                     mailaddr ? mailaddr : "???", argc,
                     (argc == 1) ? "" : "s") == EOF)
         {
                     (void) mlfi_cleanup(ctx, FALSE);
                     return SMFIS_TEMPFAIL;
         }

          /* continue processing */
          return SMFIS_CONTINUE;
}

sfsistat
mlfi_envrcpt(ctx, argv)
            SMFICTX *ctx;
            char **argv;
{
           struct mlfiPriv *priv = MLFIPRIV;
           char *rcptaddr = smfi_getsymval(ctx, "{rcpt_addr}");
           int argc = 0;

           /* count the arguments */
           while (*argv++ != NULL)
                  ++argc;

           /* log this recipient */
           if (reject != NULL && rcptaddr != NULL &&
              (strcasecmp(rcptaddr, reject) == 0))
           {
            if (fprintf(priv->mlfi_fp, "RCPT %s -- REJECTED\n",
                        rcptaddr) == EOF)
              {
                        (void) mlfi_cleanup(ctx, FALSE);
                        return SMFIS_TEMPFAIL;
               }
              return SMFIS_REJECT;
           }
         if (fprintf(priv->mlfi_fp, "RCPT %s (%d argument%s)\n",
```

```
                    rcptaddr ? rcptaddr : "???", argc,
                    (argc == 1) ? "" : "s") == EOF)
            {
                  (void) mlfi_cleanup(ctx, FALSE);
                   return SMFIS_TEMPFAIL;
             }

             /* continue processing */
             return SMFIS_CONTINUE;
}

sfsistat
mlfi_header(ctx, headerf, headerv)
            SMFICTX *ctx;
            char *headerf;
            unsigned char *headerv;
{
              /* write the header to the log file */
     if (fprintf(MLFIPRIV->mlfi_fp, "%s: %s\n", headerf,\
         headerv) == EOF)
     {
         (void) mlfi_cleanup(ctx, FALSE);
         return SMFIS_TEMPFAIL;
     }

     /* continue processing */
     return SMFIS_CONTINUE;
}

sfsistat
mlfi_eoh(ctx)
         SMFICTX *ctx;
{
   /* output the blank line between the header and the body */
   if (fprintf(MLFIPRIV->mlfi_fp, "\n") == EOF)
   {
           (void) mlfi_cleanup(ctx, FALSE);
           return SMFIS_TEMPFAIL;
   }

    /* continue processing */
    return SMFIS_CONTINUE;
}

sfsistat
mlfi_body(ctx, bodyp, bodylen)
```

```
            SMFICTX *ctx;
            unsigned char *bodyp;
            size_t bodylen;
{
             struct mlfiPriv *priv = MLFIPRIV;

             /* output body block to log file */
             if (fwrite(bodyp, bodylen, 1, priv->mlfi_fp) != 1)
             {
                  /* write failed */
                 fprintf(stderr, "Couldn't write file %s: %s\n",
                        priv->mlfi_fname, strerror(errno));
                 (void) mlfi_cleanup(ctx, FALSE);
                 return SMFIS_TEMPFAIL;
             }

             /* continue processing */
             return SMFIS_CONTINUE;
}

sfsistat
mlfi_eom(ctx)
          SMFICTX *ctx;
{
            bool ok = TRUE;

            /* change recipients, if requested */
            if (add != NULL)
                   ok = (smfi_addrcpt(ctx, add) == MI_SUCCESS);
            return mlfi_cleanup(ctx, ok);
}

sfsistat
mlfi_abort(ctx)
             SMFICTX *ctx;
{
             return mlfi_cleanup(ctx, FALSE);
}

sfsistat
mlfi_cleanup(ctx, ok)
          SMFICTX *ctx;
          bool ok;
{
          sfsistat rstat = SMFIS_CONTINUE;
          struct mlfiPriv *priv = MLFIPRIV;
```

```
        char *p;
        char host[512];
        char hbuf[1024];

        if (priv == NULL)
                    return rstat;

         /* close the archive file */
  if (priv->mlfi_fp != NULL && fclose(priv->mlfi_fp) == EOF)
   {
     /* failed; we have to wait until later */
     fprintf(stderr, "Couldn't close archive file %s: %s\n",
             priv->mlfi_fname, strerror(errno));
     rstat = SMFIS_TEMPFAIL;
     (void) unlink(priv->mlfi_fname);
   }
   else if (ok)
   {
   /* add a header to the message announcing our presence */
   if (gethostname(host, sizeof host) < 0)
      snprintf(host, sizeof host, "localhost");
   p = strrchr(priv->mlfi_fname, '/');
   if (p == NULL)
            p = priv->mlfi_fname;
   else
            p++;
    snprintf(hbuf, sizeof hbuf, "%s@%s", p, host);
  if (smfi_addheader(ctx, "X-Archived", hbuf) != MI_SUCCESS)
  {
       /* failed; we have to wait until later */
       fprintf(stderr, "Couldn't add header: X-Archived:
                                              %s\n", hbuf);
      ok = FALSE;
      rstat = SMFIS_TEMPFAIL;
      (void) unlink(priv->mlfi_fname);
  }
}
else
{
       /* message was aborted -- delete the archive file */
       fprintf(stderr, "Message aborted.  Removing %s\n",
                      priv->mlfi_fname);
       rstat = SMFIS_TEMPFAIL;
       (void) unlink(priv->mlfi_fname);
}
```

```
        /* release private memory */
        if (priv->mlfi_fname != NULL)
                free(priv->mlfi_fname);

        /* return status */
        return rstat;
}

sfsistat
mlfi_close(ctx)
        SMFICTX *ctx;
{
        struct mlfiPriv *priv = MLFIPRIV;

        if (priv == NULL)
                return SMFIS_CONTINUE;
        if (priv->mlfi_connectfrom != NULL)
                free(priv->mlfi_connectfrom);
        if (priv->mlfi_helofrom != NULL)
                free(priv->mlfi_helofrom);
        free(priv);
        smfi_setpriv(ctx, NULL);
        return SMFIS_CONTINUE;
}

struct smfiDesc smfilter =
{
        "SampleFilter",/* filter name */
        SMFI_VERSION,/* version code -- do not change */
        SMFIF_ADDHDRS|SMFIF_ADDRCPT,
                                /* flags */
        mlfi_connect,/* connection info filter */
        mlfi_helo,/* SMTP HELO command filter */
        mlfi_envfrom,/* envelope sender filter */
        mlfi_envrcpt,/* envelope recipient filter */
        mlfi_header,/* header filter */
        mlfi_eoh,/* end of header */
        mlfi_body,/* body block filter */
        mlfi_eom,/* end of message */
        mlfi_abort,/* message aborted */
        mlfi_close,/* connection cleanup */
};

static void
usage(prog)
        char *prog;
```

```
{
        fprintf(stderr,
                "Usage: %s -p socket-addr [-t timeout] [-r
                    reject-addr] [-a add-addr]\n", prog);
}

int
main(argc, argv)
        int argc;
        char **argv;
{
        bool setconn = FALSE;
        int c;
        const char *args = "p:t:r:a:h";
        extern char *optarg;

        /* Process command line options */
        while ((c = getopt(argc, argv, args)) != -1)
        {
                switch (c)
                {
                   case 'p':
                       if (optarg == NULL || *optarg == '\0')
                       {
                            (void) fprintf(stderr, "Illegal
                              conn: %s\n", optarg);
                          exit(EX_USAGE);

                       if (smfi_setconn(optarg)==MI_FAILURE)
                       {
                          (void) fprintf(stderr,
                                    "smfi_setconn failed\n");
                           exit(EX_SOFTWARE);
                        }

                /*
                **  If we're using a local socket, make sure it
                **  doesn't already exist.  Don't ever run this
                **  code as root!!
                */

                if (strncasecmp(optarg, "unix:", 5) == 0)
                                unlink(optarg + 5);
                else if (strncasecmp(optarg, "local:", 6) == 0)
                               unlink(optarg + 6);
                setconn = TRUE;
```

```
                break;

            case 't':
               if (optarg == NULL || *optarg == '\0')
               {
                    (void) fprintf(stderr, "Illegal
                           timeout: %s\n", optarg);
                exit(EX_USAGE);
               }
                 if (smfi_settimeout(atoi(optarg)) ==
                                              MI_FAILURE)
               {
                  (void) fprintf(stderr,
                              "smfi_settimeout failed\n");
                exit(EX_SOFTWARE);
               }
              break;

            case 'r':
                 if (optarg == NULL)
                 {
                      (void) fprintf(stderr,
                      "Illegal reject rcpt: %s\n",optarg);
                      exit(EX_USAGE);
                 }
                 reject = optarg;
                 break;

            case 'a':
                 if (optarg == NULL)
                 {
                    (void) fprintf(stderr,
                      "Illegal add rcpt: %s\n", optarg);
                    exit(EX_USAGE);
                 }
                 add = optarg;
                 smfilter.xxfi_flags |= SMFIF_ADDRCPT;
                 break;

            case 'h':
            default:
                  usage(argv[0]);
                  exit(EX_USAGE);
           }
      }
      if (!setconn)
```

```
                {
                        fprintf(stderr, "%s: Missing required -p
                                                argument\n", argv[0]);
                        usage(argv[0]);
                        exit(EX_USAGE);
                }
                if (smfi_register(smfilter) == MI_FAILURE)
                {
                        fprintf(stderr, "smfi_register failed\n");
                        exit(EX_UNAVAILABLE);
                }
                return smfi_main();
}

/* eof */
```

This sample program logs each message to a separate temporary file, adds a recipient given with the `-a` flag, and rejects a disallowed recipient address given with the `-r` flag. The sample program recognizes the following options:

`-p` *port*	Specifies the port through which Sendmail connects to the filter.
`-t` *sec*	Specifies the timeout value.
`-r` *addr*	Specifies a recipient to reject.
`-a` *addr*	Specifies a recipient to add.