Mailutils, SDK.

version 0.2, 16 December 2002

Alain Magloire et al.

Copyright © 1999, 2000, 2001, 2002 Free Software Foundation, Inc.

Published by the Free Software Foundation,
59 Temple Place - Suite 330,
Boston, MA 02111-1307, USA

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Foundation.

Short Contents

1 Introduction 1
2 Framework 2
3 Authentication Library 36
4 Mailutils to Scheme interface 41
5 Sieve Library 45
6 Programs 71
7 Reporting Bugs 121
8 Getting news about GNU mailutils 121
9 Acknowledgement 121
Function Index 122
Program Index 126
Concept Index 127

1 Introduction

GNU Mailutils offers a general purpose library whose aim is to provide a rich set of functions for accessing different mailbox formats and mailers.

1.1 References

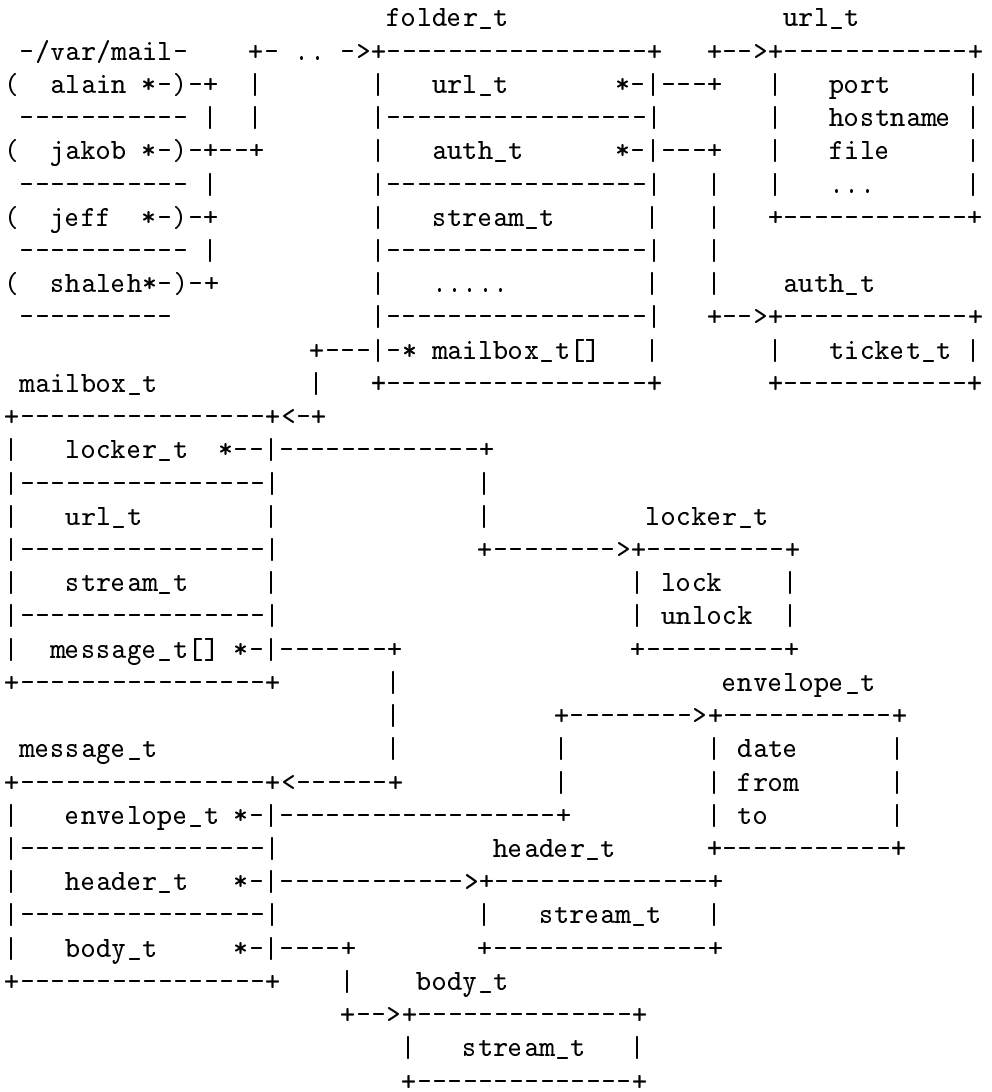
For more information on,

- SMTP
  - *RFC 2821: Simple Mail Transfer Protocol*
  - *RFC 2368: The mailto URL scheme*
- POP3
  - *RFC 1939: Post Office Protocol - Version 3*
  - *RFC 1734: POP3 AUTHentication command*
  - *RFC 1957: Some Observations on Implementations of the Post Office Protocol (POP3)*
  - *RFC 2449: POP3 Extension Mechanism*
  - *RFC 2384: POP URL Scheme*
- IMAP4
  - *RFC 2060: INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1*
  - *RFC 2088: IMAP4 non-synchronizing literals*
  - *RFC 2193: IMAP4 Mailbox Referrals*
  - *RFC 2221: IMAP4 Login Referrals*
  - *RFC 2342: IMAP4 Namespace*
  - *RFC 2192: IMAP URL Scheme*
  - *RFC 1731: IMAP4 Authentication Mechanisms*
  - *RFC 2245: Anonymous SASL Mechanism*
- message formats
  - *RFC 2822: Internet Message Format*
  - *RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies*
  - *RFC 2046: Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*
  - *RFC 2047: Multipurpose Internet Mail Extensions (MIME) Part Three: Message Header Extensions for Non-ASCII Text*
  - *RFC 2049: Multipurpose Internet Mail Extensions (MIME) Part Five: Conformance Criteria and Examples*
  - *RFC 2111: Content-ID and Message-ID Uniform Resource Locators*
- miscellaneous related topics
  - *RFC 3028: Sieve: A Mail Filtering Language*
  - *RFC 2298: An Extensible Message Format for Message Disposition Notifications*

– *RFC 1738: Uniform Resource Locators (URL)*
– *Internet Email Protocols: A Developer's Guide, by Kevin Johnson*

2 Framework

Wherever the mail is and whatever format it is stored in, it is operated upon using the same set of functions. To unified the C API, GNU Mailutils offers a heteroclite set of objects that work in aggregation to do operations on emails. Each object does a specific task and delegates non-related tasks to others. The object comes alive by specifying a *URL* parameter when created, it will indicate the storage format or protocol (POP3, IMAP4, MH, MAILDIR, etc ..).



As an example, here is a simplified version of `from` command. It lists the '`From`' and '`Subject`' headers of every mail in a mailbox.

```
/* sfrom, Simple From */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

#include <mailutils/mailutils.h>

int
main (int argc, const char **argv)
{
  char *from;
  char *subject;
  mailbox_t mbox;
  int status;
  size_t msgno, total = 0;

  /* Register the type of mailbox. IMAP4, POP3 and local format  */
  {
    list_t registrar;
    registrar_get_list (&registrar);
    list_append (registrar, imap_record);
    list_append (registrar, path_record);
    list_append (registrar, pop_record);
  }

  status = mailbox_create_default (&mbox, argv[1]);

  if (status != 0)
    {
      fprintf (stderr, "mailbox_create: %s\n", mu_errstring (status));
      exit (EXIT_FAILURE);
    }

  status = mailbox_open (mbox, MU_STREAM_READ);
  if (status != 0)
    {
      fprintf (stderr, "mailbox_open: %s\n", mu_errstring (status));
      exit (EXIT_FAILURE);
    }

  mailbox_messages_count (mbox, &total);

  for (msgno = 1; msgno <= total; msgno++)
    {
      message_t msg;
      header_t hdr;
```

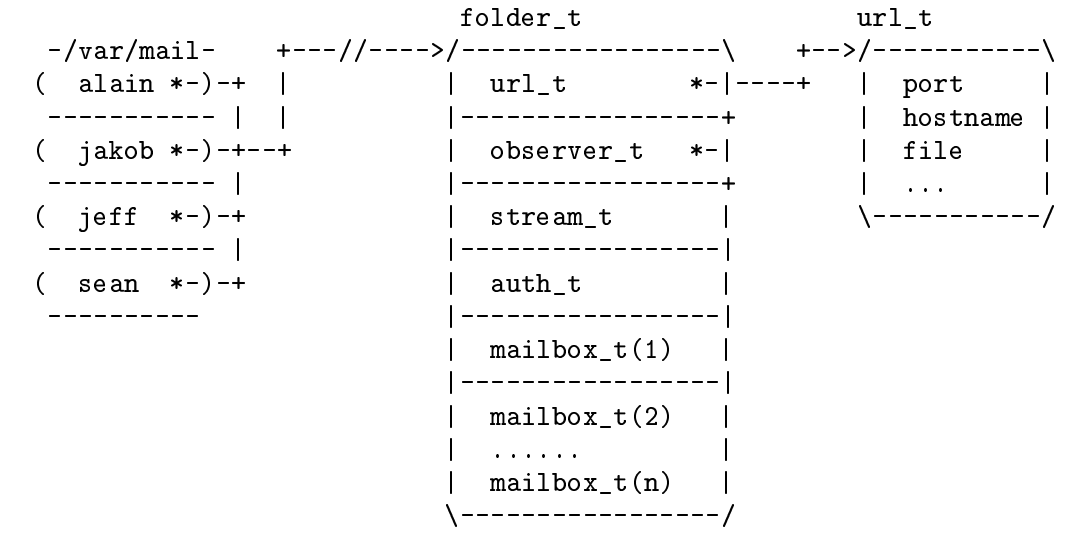
```
      if ((status = mailbox_get_message (mbox, msgno, &msg)) != 0
          || (status = message_get_header (msg, &hdr)) != 0)
        {
          fprintf (stderr, "Error message: %s\n",
                   mu_errstring (status));
          exit (EXIT_FAILURE);
        }

      if (header_aget_value (hdr, MU_HEADER_FROM, &from))
        from = strdup ("(NO FROM)");

      if (header_aget_value (hdr, MU_HEADER_SUBJECT, &subject))
        subject = strdup("(NO SUBJECT)");

      printf ("%s\t%s\n", from, subject);
      free (from);
      free (subject);
    }

  mailbox_close (mbox);
  mailbox_destroy (&mbox);
  return 0;
}
```

Here is a sample output produced by this program:

```
% MAIL=pop://alain@localhost ./sfrom
Passwd: xxxx
Jim Meyering <meyering@foo.org>      fetish(shellutils) beta
François Pinard <pinard@bar.org> recode new alpha
...
```

2.1 Folder

```
/* Prefix folder_ is reserve */
#include <mailutils/folder.h>
```



int **folder_create** (folder_t *, const char **url*) [Function]

void **folder_destroy** (folder_t *) [Function]

int **folder_open** (folder_t, int *flag*) [Function]

int **folder_close** (folder_t) [Function]

int **folder_delete** (folder_t, const char **mailbox*) [Function]

int **folder_rename** (folder_t, const char *, const char **mailbox*) [Function]

int **folder_subscribe** (folder_t, const char **mailbox*) [Function]

int **folder_unsubscribe** (folder_t, const char **mailbox*) [Function]

int **folder_list** (folder_t, const char **ref*, const char **wcard*, iterator_t *) [Function]

int **folder_lsub** (folder_t, const char **ref*, const char **wcar*, iterator_t *) [Function]

int **folder_get_stream** (folder_t, stream_t *) [Function]

int **folder_set_stream** (folder_t, stream_t) [Function]

int **folder_get_observable** (folder_t, observable_t *) [Function]

int **folder_get_debug** (folder_t, debug_t *) [Function]

int **folder_set_debug** (folder_t, debug_t) [Function]

int **folder_get_authority** (folder_t, authority_t *) [Function]

`int` **folder_set_authority** (`folder_t, authority_t`) [Function]

`int` **folder_get_url** (`folder_t, url_t *`) [Function]

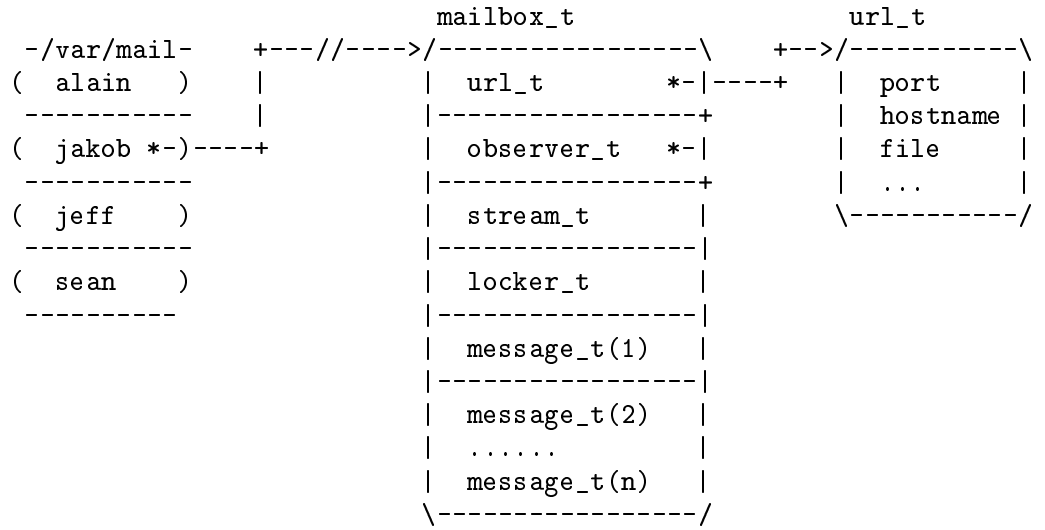
`int` **folder_set_url** (`folder_t, url_t`) [Function]

2.2 Mailbox

```
/* Prefix mailbox_ is reserved */
#include <mailutils/mailbox.h>
```

mailbox_t [Data Type]

The `mailbox_t` object is used to hold information and it is an opaque data structure to the user. Functions are provided to retrieve information from the data structure.



`int` **mailbox_append_message** (`mailbox_t` *mbox*, `message_t` *message*) [Function]

The *message* is appended to the mailbox *mbox*.

The return value is `0` on success and a code number on error conditions:

`MU_ERROR_INVALID_PARAMETER`
: *mbox* is null or *message* is invalid.

`int` **mailbox_close** (`mailbox_t` *mbox*) [Function]

The stream attach to *mbox* is closed.

The return value is `0` on success and a code number on error conditions:

`MU_ERROR_INVALID_PARAMETER`
: *mbox* is null.

`int` **mailbox_create** (`mailbox_t` *pmbox, `const char` *name) [Function]
The function `mailbox_create` allocates and initializes *pmbox*. The concrete mailbox type instantiate is based on the scheme of the url *name*.

The return value is `0` on success and a code number on error conditions:

`MU_ERR_OUT_PTR_NULL`
The pointer *pmbox* supplied is NULL.

`MU_ERR_NO_HANDLER`
The url *name* supplied is invalid or not supported.

`EINVAL`

`ENOMEM` Not enough memory to allocate resources.

`int` **mailbox_create_default** (`mailbox_t` *pmbox, `const char` *name) [Function]

Create a mailbox with `mailbox_create ()` based on the environment variable *$MAIL* or the string formed by *_PATH_MAILDIR/user*" or *$LOGNAME* if *user* is null,

`void` **mailbox_destroy** (`mailbox_t` *pmbox) [Function]
Destroys and releases resources held by *pmbox*.

`int` **mailbox_expunge** (`mailbox_t` *mbox*) [Function]
All messages marked for deletion are removed.

The return value is `0` on success and a code number on error conditions:

`MU_ERROR_INVALID_PARAMETER`
mbox is null.

`int` **mailbox_get_folder** (`mailbox_t` *mbox*, `folder_t` **folder*) [Function]
Get the *folder*.

The return value is `0` on success and a code number on error conditions:

`MU_ERROR_INVALID_PARAMETER`
mbox is null.

`int` **mailbox_get_debug** (`mailbox_t` *mbox*, `debug_t` **debug*) [Function]
Get a debug object. The return value is `0` on success and a code number on error conditions:

`MU_ERROR_INVALID_PARAMETER`
mbox is null.

`ENOMEM`

`int` **mailbox_get_locker** (`mailbox_t` *mbox*, `locker_t` **plocker*) [Function]
Return the *locker* object.

The return value is `0` on success and a code number on error conditions:

`MU_ERROR_INVALID_PARAMETER`
mbox is null.

`int` **mailbox_get_message** (`mailbox_t` *mbox*, `size_t` *msgno*, `message_t *`*pmessage*) [Function]

Retrieve message number *msgno*, *pmessage* is allocated and initialized.

The return value is `0` on success and a code number on error conditions:

`MU_ERROR_INVALID_PARAMETER`
: *mbox* is null or *msgno* is invalid.

`ENOMEM`
: Not enough memory.

`int` **mailbox_get_observable** (`mailbox_t mbox` *mbox*, `observable_t*`*observable*) [Function]

Get the observable object.

The return value is `0` on success and a code number on error conditions:

`MU_ERROR_INVALID_PARAMETER`
: *mbox* is null.

`ENOMEM`
: Not enough memory.

`int` **mailbox_get_property** (`mailbox_t` *mbox*, `property_t *`*property*) [Function]

Get the property object. The return value is `0` on success and a code number on error conditions:

`MU_ERROR_INVALID_PARAMETER`
: *mbox* is null.

`ENOMEM`

`int` **mailbox_get_size** (`mailbox_t` *mbox*, `off_t *`*psize*) [Function]

Gives the *mbox* size.

The return value is `0` on success and a code number on error conditions:

`MU_ERROR_INVALID_PARAMETER`
: *mbox* is null.

`int` **mailbox_get_stream** (`mailbox_t` *mbox*, `stream_t *`*pstream*) [Function]

The mailbox stream is put in *pstream*.

The return value is `0` on success and a code number on error conditions:

`MU_ERROR_INVALID_PARAMETER`
: *mbox* is invalid or *pstream* is NULL.

`int` **mailbox_get_ticket** (`mailbox_t` *mbox*, `ticket_t` *ticket*) [Function]

The return value is `0` on success and a code number on error conditions:

`MU_ERROR_INVALID_PARAMETER`
: *mbox* is null.

`int` **mailbox_get_url** (`mailbox_t` *mbox*, `url_t *`*purl*) [Function]

Gives the constructed *url*.

The return value is `0` on success and a code number on error conditions:

`MU_ERROR_INVALID_PARAMETER`
: *mbox* is null.

`int` **mailbox_is_modified** (`mailbox_t` *mbox*) [Function]

Check if the mailbox been modified by an external source.

The return value is `0` on success and a code number on error conditions:

`MU_ERROR_INVALID_PARAMETER`
: *mbox* is null.

`int` **mailbox_message_unseen** (`mailbox_t` *mbox*, `size_t` `*`*pnumber*); [Function]

Give the number of first unseen message in *mbox*.

The return value is `0` on success and a code number on error conditions:

`MU_ERROR_INVALID_PARAMETER`
: *mbox* is null.

`int` **mailbox_messages_count** (`mailbox_t` *mbox*, `size_t *`*pnumber*); [Function]

Give the number of messages in *mbox*.

The return value is `0` on success and a code number on error conditions:

`MU_ERROR_INVALID_PARAMETER`
: *mbox* is null.

`int` **mailbox_messages_recent** (`mailbox_t` *mbox*, `size_t` `*`*pnumber*); [Function]

Give the number of recent messages in *mbox*.

The return value is `0` on success and a code number on error conditions:

`MU_ERROR_INVALID_PARAMETER`
: *mbox* is null.

`int` **mailbox_open** (`mailbox_t` *mbox*, `int` *flag*) [Function]

A connection is open, if no stream was provided, a stream is created based on the *mbox* type. The *flag* can be OR'ed. See `stream_create` for *flag*'s description.

The return value is `0` on success and a code number on error conditions:

`EAGAIN`
`EINPROGRESS`
: Operation in progress.

`EBUSY`
: Resource busy.

`MU_ERROR_INVALID_PARAMETER`
mbox is null or flag is invalid.

`ENOMEM` Not enough memory.

`int` **mailbox_scan** (`mailbox_t` *mbox*, `size_t` *msgno*, `size_t` **pcount*); [Function]
Scan the mailbox for new messages starting at message *msgno*.

The return value is `0` on success and a code number on error conditions:

`MU_ERROR_INVALID_PARAMETER`
mbox is null.

`ENOMEM` Not enough memory.

`int` **mailbox_set_locker** (`mailbox_t` *mbox*, `locker_t` *locker*) [Function]
Set the type of locking done by the *mbox*.

The return value is `0` on success and a code number on error conditions:

`MU_ERROR_INVALID_PARAMETER`
mbox is null.

`int` **mailbox_set_stream** (`mailbox_t` *mbox*, `stream_t` *stream*) [Function]
Set the *stream* connection to use for the mailbox.

The return value is `0` on success and a code number on error conditions:

`MU_ERROR_INVALID_PARAMETER`
mbox or *stream* is NULL.

`int` **mailbox_set_ticket** (`mailbox_t` *mbox*, `ticket_t` *ticket*) [Function]
The *ticket* will be set on the `auth_t` object on creation.

The return value is `0` on success and a code number on error conditions:

`MU_ERROR_INVALID_PARAMETER`
mbox is null.

`int` **mailbox_uidnext** (`mailbox_t` *mbox*, `size_t` **pnumber*); [Function]
Give the next predicted uid for *mbox*.

The return value is `0` on success and a code number on error conditions:

`MU_ERROR_INVALID_PARAMETER`
mbox is null.

`int` **mailbox_uidvalidity** (`mailbox_t` *mbox*, `size_t` **pnumber*); [Function]
Give the uid validity of *mbox*.

The return value is `0` on success and a code number on error conditions:

`MU_ERROR_INVALID_PARAMETER`
mbox is null.

2.3 Mailer

```
/* Prefix mailer_ is reserved */
#include <mailutils/mailer.h>
```

The API is still changing.

`int` **mailer_create** (`mailer_t *, const char *`) [Function]

`void` **mailer_destroy** (`mailer_t *`) [Function]

`int` **mailer_open** (`mailer_t, int flags`) [Function]

`int` **mailer_close** (`mailer_t`) [Function]

`int` **mailer_send_message** (`mailer_t` *mailer*, `message_t` *msg*, `address_t` *from*, `address_t` *to*); [Function]

If from is not *NULL*, it must contain a single fully qualified RFC2822 email address which will be used as the envelope from address. This is the address to which delivery status notifications are sent, so it never matters what it is set to until it REALLY matters. This is equivalent to sendmail's *-f* flag.

The default for *from* is provided by the specific mailer.

If to is not *NULL*, then the message will be sent to the list of addresses that it specifies.

The default for *to* is to use the contents of the standard "To:", "Cc:", and "Bcc:" fields, this is equivalent to sendmail's *-t* flag.

Note: the previous implementation of mailer_send_message() was equivalent to having both *from* and *to* be *NULL*.

`int` **mailer_get_property** (`mailer_t, property_t *`) [Function]

`int` **mailer_get_stream** (`mailer_t, stream_t *`) [Function]

`int` **mailer_set_stream** (`mailer_t, stream_t`) [Function]

`int` **mailer_get_debug** (`mailer_t, debug_t *`) [Function]

`int` **mailer_set_debug** (`mailer_t, debug_t`) [Function]

`int` **mailer_get_observable** (`mailer_t, observable_t *`) [Function]

`int` **mailer_get_url** (`mailer_t, url_t *`) [Function]

Some usage notes.

Some possible use cases the API must support are:

- original submission

1 - fill in header addresses

2 - mailer_send_message(mailer, msg, NULL, NULL);

- from will be filled in if missing,

- bcc's will be deleted before delivery to a non-bcc address,

- message-id and date will be added, if missing,

- a to: or apparently-to: will be added if non is present (for RFC compliance)

- MTA-style .forward (and sieve-style redirect)

1 - get the envelope from of the message to be forwarded

2 - mailer_send_message(mailer, msg, from, to)

- MUA-style bounce

1 - add Resent-[to,from,....]

2 - mailer_send_message(mailer, msg, NULL, to)

- DSN "bounce"

1 - compose DSN

2 - mailer_deliver(mailer, msg, address_t("<>"), to)

Don't want mail loops, so the null but valid SMTP address of <> is the envelope from.

The sendmail mailer.

/sbin/sendmail isn't always sendmail... sometimes its a sendmail-compatible wrapper, so assume /sbin/sendmail understands only a recipient list, -f and -oi, these seem to be pretty basic. Cross fingers.

Pipe to "/sbin/sendmail -oi [-f from] [to...]", supplying -f if there was a from, and supplying the recipient list from the to (if there is no recipient list, assume it will read the message contents for the recipients).

Note: since the stdout and stderr of sendmail is closed, we have no way of ever giving feedback on failure. Also, what should the return code be from mailer_send_message() when sendmail returns 1? 1 maps to EPERM, which is less than descriptive!

The SMTP mailer.

This mailer does NOT canonicalize the message. This must be done before sending the message, or it may be assumed that the MTA will do so.

It does blind out the Bcc: header before sending, though.

Note: mutt always puts the recipient addresses on the command line, even bcc ones, do we strip the bcc before forwarding with SMTP?

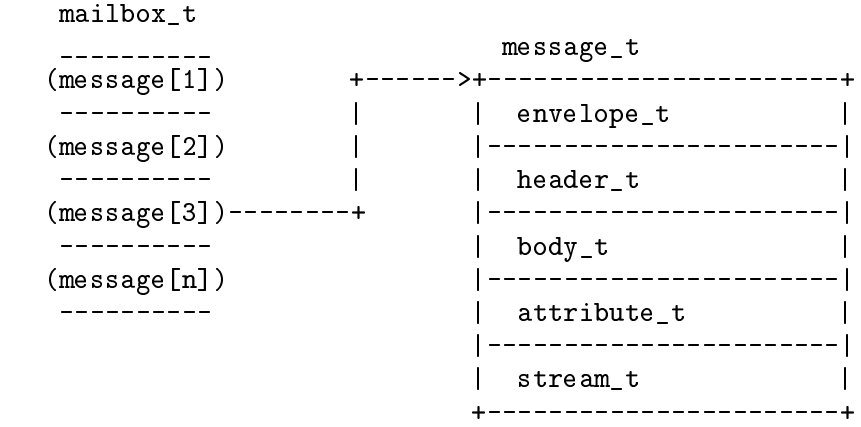
Non-RFC822 addresses.

An address that has no domain is not and RFC822 email address. What do I do with them? Should the user of the API be responsible for determining what is mean by email to "john" means? Or should the be able to configure sendmail to decide globally what this means. If so, we can pass the address to sendmail, but we have to decide for SMTP! So, right now these addresses are rejected. This could be changed.

2.4 Message

```
#include <mailutils/message.h>
/* Prefix message_ is reserve */
```

The `message_t` object is a convenient way to manipulate messages. It encapsulates the `envelope_t`, the `header_t` and the `body_t`.



`void` **message_create** (`message_t` *_msg_, `void` *_owner_) [Function]

`void` **message_destroy** (`message_t` *_msg_, `void` *_owner_) [Function]
The resources allocate for _msg_ are freed.

`int` **message_get_header** (`message_t` _msg_, `header_t` *_pheader_) [Function]
Retrieve _msg_ header.

`int` **message_set_header** (`message_t` _msg_, `header_t` _header_, `void` *_owner_) [Function]

`int` **message_get_body** (`message_t` _msg_, `body_t` *_pbody_) [Function]

`int` **message_set_body** (`message_t` _msg_, `body_t` _body_, `void` *_owner_) [Function]

`int` **message_is_multipart** (`message_t` _msg_, `int` *_pmulti_) [Function]
Set *_pmulti_ to non-zero value if _msg_ is multi-part.

`int` **message_get_num_parts** (`message_t` _msg_, `size_t` *`nparts`) [Function]

`int` **message_get_part** (`message_t` *msg*, `size_t` `part`, `message_t` `*msg`) [Function]

`int` **message_get_stream** (`message_t` *msg*, `stream_t` `*`*pstream*) [Function]

`int` **message_set_stream** (`message_t` *msg*, `stream_t` *stream*,`void` `*`*owner*) [Function]

`int` **message_get_attribute** (`message_t` *msg*, `attribute_t` `*`*pattribute*) [Function]

`int` **message_set_attribute** (`message_t` *msg*, `attribute_t` *attribute*, `void` `*owner`) [Function]

`int` **message_get_envelope** (`message_t` *msg*, `envelope_t` `*penvelope`) [Function]

`int` **message_set_envelope** (`message_t` *msg*, `envelope_t` `envelope`, `void` `*`*owner*) [Function]

`int` **message_get_uid** (`message_t` *msg*, `size_t` `*`*uid*) [Function]

`int` **message_get_uidl** (`message_t` *msg*, `char` `*`*buffer*, `size_t` *buflen*, `size_t` `*`*pwriten*) [Function]

`int` **message_set_uidl** (`message_t` *msg*, `int` (`*`*_get_uidl*)(`message_t`, `char *`, `size_t`, `size_t *`), `void` `*`*owner*) [Function]

`int` **message_get_observable** (`message_t` *msg*, `observable_t` `*`*observable*) [Function]

`int` **message_create_attachment** (`const char` `*`*content_type*, `const char` `*`*encoding*, `const char` `*`*filename*, `message_t` `*`*newmsg*) [Function]

`int` **message_save_attachment** (`message_t` *msg*, `const char` `*`*filename*, `void` `**`*data*) [Function]

`int` **message_encapsulate** (`message_t` *msg*, `message_t` `*`*newmsg*, `void` `**`*data*) [Function]

`int` **message_unencapsulate** (`message_t` *msg*, `message_t` `*`*newmsg*, `void` `**`*data*); [Function]

2.5 Envelope

```
/* Prefix envelope_ is reserved */
#include <mailutils/envelope.h>
```

`int` **envelope_date** (`envelope_t, char *, size_t, size_t *`); [Function]
Get the date that the message was delivered to the mailbox, in something close to ANSI ctime() format: Mon Jul 05 13:08:27 1999.

`int` **envelope_sender** (`envelope_t, char *, size_t, size_t *`); [Function]
Get the address that this message was reportedly received from. This would be the "mail from" argument if the message was delivered or received via SMTP, for example.

`int` **envelope_get_message** (`envelope_t, message_t *`); [Function]

`int` **envelope_create** (`envelope_t *, void *`); [Function]
Primarily for internal use.

`void` **envelope_destroy** (`envelope_t *, void *`); [Function]
Primarily for internal use.

`int` **envelope_set_sender** (`envelope_t, int (*_sender) __P ((envelope_t, char *, size_t, size_t*)), void *`); [Function]
Primarily for internal use. The implementation of envelope_t depends on the mailbox type, this allows the function which actually gets the sender to be set by the creator of an envelope_t.

`int` **envelope_set_date** (`envelope_t, int (*_date) __P ((envelope_t, char *, size_t, size_t *)), void *`); [Function]
Primarily for internal use. The implementation of envelope_t depends on the mailbox type, this allows the function which actually gets the date to be set by the creator of an envelope_t.

2.6 Headers

```
/* Prefix header_ is reserved */
#include <mailutils/header.h>
```

So far we plan support for RFC822 and plan for RFC1522. with RFC1522 non ASCII characters will be encoded.

`int` **header_create** (`header_t *`*hdr*`, const char *`*blurb*`, size_t` *len*`, void *`*owner*`)` [Function]
Initialize a *hdr* to a supported type. If *blurb* is not NULL, it is parsed.

`void` **header_destroy** (`header_t *`*hdr*`, void *`*owner*`)` [Function]
The resources allocated for *hdr* are freed.

`int` **header_set_value** (`header_t` *hdr*, `const char *`*fn*, `const char *`*fv*, `size_t` `n`, `int` *replace*) [Function]

Set the field-name *fn* to field-value *fv* of size *n* in *hdr*. If *replace* is non-zero the initial value is replaced, if zero it is appended.

Some basic macros are already provided for rfc822.

`MU_HEADER_RETURN_PATH`
: Return-Path

`MU_HEADER_RECEIVED`
: Received

`MU_HEADER_DATE`
: Date

`MU_HEADER_FROM`
: From

`MU_HEADER_RESENT_FROM`
: Resent-From

`MU_HEADER_SUBJECT`
: Subject

`MU_HEADER_SENDER`
: Sender

`MU_HEADER_RESENT_SENDER`
: Resent-SENDER

`MU_HEADER_TO`
: To

`MU_HEADER_RESENT_TO`
: Resent-To

`MU_HEADER_CC`
: Cc

`MU_HEADER_RESENT_CC`
: Resent-Cc

`MU_HEADER_BCC`
: Bcc

`MU_HEADER_RESENT_BCC`
: Resent-Bcc

`MU_HEADER_REPLY_TO`
: Reply-To

`MU_HEADER_RESENT_REPLY_TO`
: Resent-Reply-To

`MU_HEADER_MESSAGE_ID`
: Message-ID

`MU_HEADER_RESENT_MESSAGE_ID`
: Resent-Message-ID

`MU_HEADER_IN_REPLY_TO`
: In-Reply-To

`MU_HEADER_ENCRYPTED`
: Encrypted

`MU_HEADER_PRECEDENCE`
: Precedence

`MU_HEADER_STATUS`
: Status

`MU_HEADER_CONTENT_LENGTH`
: Content-Length

`MU_HEADER_CONTENT_TYPE`
: Content-Type

`MU_HEADER_MIME_VERSION`
: MIME-Version

`int` **header_get_value** (`header_t` *hdr*, `const char *`*fn*, `char *`*fv*, `size_t` *len*, `size_t *`*n*) [Function]
Value of field-name *fn* is returned in buffer *fv* of size *len*. The number of bytes written is put in *n*.

`int` **header_aget_value** (`header_t` *hdr*, `const char *`*fn*, `char **`*fv*) [Function]
The value is allocated.

`int` **header_get_stream** (`header_t` *hdr*, `stream_t *`*pstream*) [Function]

`int` **header_set_size** (`header_t` *hdr*, `size_t *`*size*) [Function]

`int` **header_set_lines** (`header_t` *hdr*, `size_t *`*lpines*) [Function]

2.7 Body

```
/* Prefix body_ is reserved */
#include <mailutils/body.h>
```

`int` **body_create** (`body_t *`*body*, `void *`*owner*) [Function]
Initialize an object *bdy*.

`void` **body_destroy** (`body_t *`*pbody*) [Function]
The resources allocate are release.

`int` **body_get_stream** (`body_t` *body*, `stream_t *`*pstream*) [Function]

`int` **body_set_stream** (`body_t` *body*, `stream_t` *stream*, `void *`*owner*) [Function]

`int` **body_get_filename** `__P` ((`body_t` *body*, `char *`*buffer*, `size_t`*buflen*, `size_t *`*pwriten*) [Function]

`int` **body_set_filename** (`body_t` *body*, `const char*`*buffer*) [Function]

`int` **body_size** (`body_t` *body*, `size_t*`*psize*) [Function]

`int` **body_lines** (`body_t` *body*, `size_t *`*plines*) [Function]

2.8 Attribute

```
/* Prefix attribute_ is reserved */
#include <mailutils/attribute.h>
```

`int` **attribute_create** (`attribute_t *`*pattribute*) [Function]

`void` **attribute_destroy** (`attribute_t *`*pattribute*) [Function]

`int` **attribute_is_seen** (`attribute_t` *attribute*) [Function]

`int` **attribute_is_answered** (`attribute_t` *attribute*) [Function]

`int` **attribute_is_flagged** (`attribute_t` *attribute*) [Function]

`int` **attribute_is_deleted** (`attribute_t` *attribute*) [Function]

`int` **attribute_is_draft** (`attribute_t` *attribute*) [Function]

`int` **attribute_is_recent** (`attribute_t` *attribute*) [Function]

`int` **attribute_is_read** (`attribute_t` *attribute*) [Function]

`int` **attribute_set_seen** (`attribute_t` *attribute*) [Function]

`int` **attribute_set_answered** (`attribute_t` *attribute*) [Function]

`int` **attribute_set_flagged** (`attribute_t` *attribute*) [Function]

`int` **attribute_set_deleted** (`attribute_t` *attribute*) [Function]

`int` **attribute_set_draft** (`attribute_t` *attribute*) [Function]

`int` **attribute_set_recent** (`attribute_t` *attribute*) [Function]

`int` **attribute_set_read** (`attribute_t` *attribute*) [Function]

`int` **attribute_unset_seen** (`attribute_t` *attribute*) [Function]

`int` **attribute_unset_answered** (`attribute_t` *attribute*) [Function]

`int` **attribute_unset_flagged** (`attribute_t` *attribute*) [Function]

`int` **attribute_unset_deleted** (`attribute_t` *attribute*) [Function]

`int` **attribute_unset_draft** (`attribute_t` *attribute*) [Function]

`int` **attribute_unset_recent** (`attribute_t` *attribute*) [Function]

`int` **attribute_unset_read** (`attribute_t` *attribute*) [Function]

`int` **attribute_is_equal** (`attribute_t` *att1*, `attribute_t` *att2*) [Function]

`int` **attribute_copy** (`attribute_t` *dst*, `attribute_t` *src*) [Function]

`int` **string_to_attribute** (`const char *`*buf*, `attribute_t *`*pattr*) [Function]

`int` **attribute_to_string** (`attribute_t` *attr*, `char *`*buf*, `size_t` *len*, `size_t *`*pwriten*) [Function]

2.9 Stream

```
#include <mailutils/stream.h>
```

These generic flags are interpreted as appropriate to the specific streams.

`MU_STREAM_READ`
: The stream is open read only.

`MU_STREAM_WRITE`
: The stream is open write only.

`MU_STREAM_RDWR`
: The stream is open read and write.

`MU_STREAM_APPEND`
: The stream is open in append mode for writing.

`MU_STREAM_CREAT`
: The stream open will create the underlying resource (such as a file) if it doesn't exist already.

`MU_STREAM_NONBLOCK`
: The stream is set non blocking.

`MU_STREAM_NO_CHECK`
: Stream is destroyed without checking for the owner.

`MU_STREAM_NO_CLOSE`
: Stream doesn't close it's underlying resource when it is closed or destroyed.

`int` **file_stream_create** (`stream_t *`*pstream*, `const char *`*filename*, `int` *flags*) [Function]

`int` **tcp_stream_create** (`stream_t` **pstream*, `const char` **host*, `int` *port*, `int` *flags*) [Function]

`int` **mapfile_stream_create** (`stream_t` **pstream*, `const char` **filename*, `int` *flags*) [Function]

`int` **memory_stream_create** (`stream_t` **pstream*, `const char` **filename*, `int` *flags*) [Function]

`int` **encoder_stream_create** (`stream_t` **pstream*, `stream_t` *iostream*, `const char` **encoding*) [Function]

`int` **decoder_stream_create** (`stream_t` **pstream*, `stream_t` *iostream*, `const char` **encoding*) [Function]

`int` **stdio_stream_create** (`stream_t` **pstream*, `FILE*` *stdio*, `int` *flags*) [Function]

If MU_STREAM_NO_CLOSE is specified, fclose() will not be called on *stdio* when the stream is closed.

`void` **stream_destroy** (`stream_t` **pstream*, `void` **owner*) [Function]

`int` **stream_open** (`stream_t` *stream*) [Function]

`int` **stream_close** (`stream_t` *stream*) [Function]

`int` **stream_is_seekable** (`stream_t` *stream*) [Function]

`int` **stream_get_fd** (`stream_t` *stream*, `int` **pfd*) [Function]

`int` **stream_read** (`stream_t` *stream*, `char` **buffer*, `size_t` *buflen*, `off_t` *offset*, `size_t` **pwriten*) [Function]

`int` **stream_readline** (`stream_t` *stream*, `char` **buffer*, `size_t` *buflen*, `off_t` *offset*, `size_t` **pwriten*) [Function]

`int` **stream_size** (`stream_t` *stream*, `off_t` **psize*) [Function]

`int` **stream_truncate** (`stream_t` *stream*, `off_t` *size*) [Function]

`int` **stream_write** (`stream_t` *stream*, `const char` **buffer*, `size_t` *buflen*, `off_t` *offset*, `size_t` **pwriten*) [Function]

`int` **stream_setbufsiz** (`stream_t` *stream*, `size_t` *size*) [Function]

`int` **stream_flush** (`stream_t` *stream*) [Function]

These functions will typically only be useful to implementors of streams.

`int` **stream_create** (`stream_t *`*pstream*`, int` *flags*`, void *`*owner*) [Function]
Used to implement a new kind of stream.

`int` **stream_get_flags** (`stream_t` *stream*`, int *`*pflags*) [Function]

`int` **stream_get_state** (`stream_t` *stream*`, int *`*pstate*) [Function]

`MU_STREAM_STATE_OPEN`
: Last action was `stream_open`.

`MU_STREAM_STATE_READ`
: Last action was `stream_read` or `stream_readline`.

`MU_STREAM_STATE_WRITE`
: Last action was `stream_write`.

`MU_STREAM_STATE_CLOSE`
: Last action was `stream_close`.

An example using `tcp_stream_create` to make a simple web client:

```
/* This is an example program to illustrate the use of stream functions.
   It connects to a remote HTTP server and prints the contents of its
   index page */
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <unistd.h>

#include <mailutils/mailutils.h>

const char *wbuf = "GET / HTTP/1.0\r\n\r\n";
char rbuf[1024];

int
main ()
{
  int ret, off = 0, fd;
  stream_t stream;
  size_t nb;
  fd_set fds;

  ret = tcp_stream_create (&stream, "www.gnu.org", 80, MU_STREAM_NONBLOCK);
  if (ret != 0)
    {
      mu_error ( "tcp_stream_create: %s\n", mu_errstring (ret));
      exit (EXIT_FAILURE);
    }

connect_again:
```

```
  ret = stream_open (stream);
  if (ret != 0)
    {
      if (ret == EAGAIN)
        {
          ret = stream_get_fd (stream, &fd);
          if (ret != 0)
            {
              mu_error ( "stream_get_fd: %s\n", mu_errstring (ret));
              exit (EXIT_FAILURE);
            }
          FD_ZERO (&fds);
          FD_SET (fd, &fds);
          select (fd + 1, NULL, &fds, NULL, NULL);
          goto connect_again;
        }
      mu_error ( "stream_open: %s\n", mu_errstring (ret));
      exit (EXIT_FAILURE);
    }

  ret = stream_get_fd (stream, &fd);
  if (ret != 0)
    {
      mu_error ( "stream_get_fd: %s\n", mu_errstring (ret));
      exit (EXIT_FAILURE);
    }

write_again:
  ret = stream_write (stream, wbuf + off, strlen (wbuf), 0, &nb);
  if (ret != 0)
    {
      if (ret == EAGAIN)
        {
          FD_ZERO (&fds);
          FD_SET (fd, &fds);
          select (fd + 1, NULL, &fds, NULL, NULL);
          off += nb;
          goto write_again;
        }
      mu_error ( "stream_write: %s\n", mu_errstring (ret));
      exit (EXIT_FAILURE);
    }

  if (nb != strlen (wbuf))
    {
      mu_error ( "stream_write: %s\n", "nb != wbuf length");
      exit (EXIT_FAILURE);
    }
```

```
    do
      {
        ret = stream_read (stream, rbuf, sizeof (rbuf), 0, &nb);
        if (ret != 0)
          {
            if (ret == EAGAIN)
              {
                FD_ZERO (&fds);
                FD_SET (fd, &fds);
                select (fd + 1, &fds, NULL, NULL, NULL);
              }
            else
              {
                mu_error ( "stream_read: %s\n", mu_errstring (ret));
                exit (EXIT_FAILURE);
              }
          }
        write (2, rbuf, nb);
      }
    while (nb || ret == EAGAIN);

    ret = stream_close (stream);
    if (ret != 0)
      {
        mu_error ( "stream_close: %s\n", mu_errstring (ret));
        exit (EXIT_FAILURE);
      }

    stream_destroy (&stream, NULL);
    exit (EXIT_SUCCESS);
  }
```

2.10 Iterator

```
/* Prefix iterator_ is reserved */
#include <mailutils/iterator.h>
```

`int` **iterator_create** (`iterator_t *`) [Function]

`void` **iterator_destroy** (`iterator_t *`) [Function]

`int` **iterator_first** (`iterator_t`) [Function]

`int` **iterator_next** (`iterator_t`) [Function]

`int` **iterator_current** (`iterator_t`, `void **pitem`) [Function]

`int` **iterator_is_done** (`iterator_t`) [Function]

2.11 Authenticator

```
/* Prefix auth_ is reserved */
#include <mailutils/auth.h>
```

There are many ways to authenticate to a server. To be flexible the authentication process is provided by two objects `auth_t` and `ticket_t`. The `auth_t` can implement different protocol like APOP, MD5-AUTH, One Time Passwd etc .. By default if a mailbox does not understand or know how to authenticate it falls back to user/passwd authentication. The `ticket_t` is a way for Mailboxes and Mailers provide a way to authenticate when the URL does not contain enough information. The default action is to call the function `auth_authenticate` which will get the *user* and *passwd* if not set, this function can be overridden by a custom method.

`int` **auth_create** (`auth_t *`*pauth*, `void *`*owner*) [Function]

`void` **auth_destroy** (`auth_t *`*pauth*, `void *`*owner*) [Function]

`int` **auth_prologue** (`auth_t` *auth*) [Function]

`int` **auth_authenticate** (`auth_t` *auth*, `char **`*user*, `char **`*passwd*) [Function]

`int` **auth_epilogue** (`auth_t` *auth*) [Function]

A simple example of an authenticate function:

```
#include <mailutils/auth.h>
#include <stdio.h>
#include <string.h>

int
my_authenticate (auth_t auth, char **user, char **passwd)
{
  char u[128] = "";
  char p[128] = "";

  /* prompt the user name */
  printf ("User: ");
  fflush (stdout);
  fgets (u, sizeof (u), stdin);
  u[strlen (u) - 1] = '\0'; /* nuke the trailing NL */

  /* prompt the passwd */
  printf ("Passwd: "); fflush (stdout);
  echo_off ();
  fgets (p, sizeof(p), stdin);
  echo_on ();
  p[strlen (p) - 1] = '\0';

  /* duplicate */
  *user = strdup (u);
```

```
  *passwd = strdup (p);
  return 0;
}
```

2.12 Address

```
/* Prefix address_ is reserved */
#include <mailutils/address.h>
```

The Internet address format is defined in RFC 822. RFC 822 has been updated, and is now superceeded by RFC 2822, which makes some corrections and clarifications. References to RFC 822 here apply equally to RFC 2822.

The RFC 822 format is more flexible than many people realize, here is a quick summary of the syntax this parser implements, see RFC 822 for the details. "[]" pairs mean "optional", "/" means "one or the other", and double-quoted characters are literals.

```
addr-spec    = local-part "@" domain
mailbox      = addr-spec ["(" display-name ")"] /
               [display-name] "<" [route] addr-spec ">"
mailbox-list = mailbox ["," mailbox-list]
group        = display-name ":" [mailbox-list] ";"
address      = mailbox / group / unix-mbox
address-list = address ["," address-list]
```

unix-mbox is a non-standard extension meant to deal with the common practice of using user names as addresses in mail utilities. It allows addresses such as "root" to be parsed correctly. These are NOT valid internet email addresses, they must be qualified before use.

Several address functions have a set of common arguments with consistent semantics, these are described here to avoid repetition.

Since an address-list may contain multiple addresses, they are accessed by a **one-based** index number, *no*. The index is one-based because pop, imap, and other message stores commonly use one-based counts to access messages and attributes of messages.

If *len* is greater than `0` it is the length of the buffer *buf*, and as much of the component as possible will be copied into the buffer. The buffer will be null terminated.

The size of a particular component may be queried by providing `0` for the *len* of the buffer, in which case the buffer is optional. In this case, if *n* is provided **n* is assigned the length of the component string.

address_t [Data Type]

The `address_t` object is used to hold information about a parsed RFC822 address list, and is an opaque data structure to the user. Functions are provided to retrieve information about an address in the address list.

`int` **address_create** (`address_t *`*addr*`, const char *`*string*) [Function]

This function allocates and initializes *addr* by parsing the RFC822 address-list *string*.

The return value is `0` on success and a code number on error conditions:

`EINVAL` Invalid usage, usually a required argument was `null`.

`ENOMEM` Not enough memory to allocate resources.

`ENOENT` Invalid RFC822 syntax, parsing failed.

`int` **address_createv** (`address_t *`*addr*`, const char *`*sv*`, size_t` *len*) [Function]
This function allocates and initializes *addr* by parsing the array of pointers to RFC822 address-lists in *sv*. If *len* is `-1`, then *sv* must be null-terminated in the fashion of *argv*, otherwise *len* is the length of the array.

The return value is `0` on success and a code number on error conditions:

`EINVAL` Invalid usage, usually a required argument was `null`.

`ENOMEM` Not enough memory to allocate resources.

`ENOENT` Invalid RFC822 syntax, parsing failed.

`void` **address_destroy** (`address_t *`*addr*) [Function]
The *addr* is destroyed.

`int` **address_get_email** (`address_t *`*addr*`, size_t` *no*`, char*` *buf*`, size_t` *len*`, size_t*` *n*) [Function]
Accesses the *no*th email address component of the address list. This address is the plain email address, correctly quoted, suitable for using in an smtp dialog, for example, or as the address part of a contact book entry.

Note that the entry may be valid, but be a group name. In this case success is returned, but the length of the address is `0`.

The return value is `0` on success and a code number on error conditions:

`EINVAL` Invalid usage, usually a required argument was `null`.

`ENOENT` The index *no* is outside of the range of available addresses.

`int` **address_aget_email** (`address_t *`*addr*`, size_t` *no*`, char**` *bufp*) [Function]
As above, but mallocs the email address, if present,

and write a pointer to it into *bufp*. *bufp* will be `NULL` if there is no email address to return.

The return value is `0` on success and a code number on error conditions:

`EINVAL` Invalid usage, usually a required argument was `null`.

`ENOENT` The index *no* is outside of the range of available addresses.

`int` **address_get_personal** (`address_t *`*addr*`, size_t` *no*`, char*` *buf*`, size_t` *len*`, size_t*` *n*) [Function]
Accesses the display-name describing the *no*th email address. This display-name is optional, so may not be present. If it is not present, but there is an RFC822 comment after the address, that comment will be returned as the personal phrase, as this is a common usage of the comment even though it is not defined in the internet mail standard.

A group is a kind of a special case. It has a display-name, followed by an optional mailbox-list. The display-name will be allocated an address all it's own, but all the other elements (local-part, domain, etc.) will be zero-length. So "a group: ;" is valid, will have a count of 1, but address_get_email(), and all the rest, will return zero-length output.

The return value is `0` on success and a code number on error conditions:

`EINVAL` Invalid usage, usually a required argument was `null`.

`ENOENT` The index *no* is outside of the range of available addresses.

`int` **address_get_comments** (`address_t *`*addr*`, size_t` *no*`, char*` *buf*`, size_t` *len*`, size_t*` *n*) [Function]

Accesses the comments extracted while parsing the *no*th email address. These comments have no defined meaning, and are not currently collected.

The return value is `0` on success and a code number on error conditions:

`EINVAL` Invalid usage, usually a required argument was `null`.

`ENOENT` The index *no* is outside of the range of available addresses.

`int` **address_get_email** (`address_t *`*addr*`, size_t` *no*`, char*` *buf*`,` `size_t` *len*`, size_t*` *n*) [Function]

Accesses the email addr-spec extracted while parsing the *no*th email address. This will be `0` length for a unix-mbox.

The return value is `0` on success and a code number on error conditions:

`EINVAL` Invalid usage, usually a required argument was `null`.

`ENOENT` The index *no* is outside of the range of available addresses.

`int` **address_get_local_part** (`address_t *`*addr*`, size_t` *no*`, char*` *buf*`, size_t` *len*`, size_t*` *n*) [Function]

Accesses the local-part of an email addr-spec extracted while parsing the *no*th email address.

The return value is `0` on success and a code number on error conditions:

`EINVAL` Invalid usage, usually a required argument was `null`.

`ENOENT` The index *no* is outside of the range of available addresses.

`int` **address_get_domain** (`address_t *`*addr*`, size_t` *no*`, char*` *buf*`,` `size_t` *len*`, size_t*` *n*) [Function]

Accesses the domain of an email addr-spec extracted while parsing the *no*th email address. This will be `0` length for a unix-mbox.

The return value is `0` on success and a code number on error conditions:

`EINVAL` Invalid usage, usually a required argument was `null`.

`ENOENT` The index *no* is outside of the range of available addresses.

`int` **address_get_route** (`address_t` *_addr_, `size_t` _no_, `char*` _buf_, `size_t` _len_, `size_t*` _n_) [Function]

Accesses the route of an email addr-spec extracted while parsing the _no_th email address. This is a rarely used RFC822 address syntax, but is legal in SMTP as well. The entire route is returned as a string, those wishing to parse it should look at <mailutils/parse822.h>.

The return value is `0` on success and a code number on error conditions:

`EINVAL` Invalid usage, usually a required argument was `null`.

`ENOENT` The index _no_ is outside of the range of available addresses.

`int` **address_is_group** (`address_t` *_addr_, `size_t` _no_, `size_t` _len_, `int*` _yes_) [Function]

Sets *_yes_ to `1` if this address is just the name of a group, `0` otherwise. This is faster than checking if the address has a non-zero length personal, and a zero-length local_part and domain.

yes can be `null`, though that doesn't serve much purpose other than determining that _no_ refers to an address.

Currently, there is no way to determine the end of the group.

The return value is `0` on success and a code number on error conditions:

`EINVAL` Invalid usage, usually a required argument was `null`.

`ENOENT` The index _no_ is outside of the range of available addresses.

`int` **address_to_string** (`address_t` *_addr_, `char*` _buf_, `size_t` _len_, `size_t*` _n_) [Function]

Returns the entire address list as a single RFC822 formatted address list.

The return value is `0` on success and a code number on error conditions:

`EINVAL` Invalid usage, usually a required argument was `null`.

`ENOMEM` Not enough memory to allocate resources.

`int` **address_get_count** (`address_t` _addr_, `size_t*` _count_) [Function]

Returns a count of the addresses in the address list.

If _addr_ is `null`, the count is `0`. If _count_ is not `null`, the count will be written to *_count_.

The return value is `0`.

2.13 Example

```
#include <stdio.h>
#include <errno.h>
#include <string.h>

#include <mailutils/address.h>
```

```
#include <mailutils/errno.h>
#include <mailutils/mutil.h>

#define EPARSE ENOENT

static int
parse (const char *str)
{
  size_t no = 0;
  size_t pcount = 0;
  int status;
  char buf[BUFSIZ];
  address_t address = NULL;

  mu_set_user_email_domain ("localhost");

  status = address_create (&address, str);

  address_get_count (address, &pcount);

  if (status)
    {
      printf ("%s=> error %s\n\n", str, mu_errname (status));
      return 0;
    }
  else
    {
      printf ("%s=> pcount %lu\n", str, (unsigned long) pcount);
    }

  for (no = 1; no <= pcount; no++)
    {
      size_t got = 0;
      int isgroup;

      address_is_group (address, no, &isgroup);

      printf ("%lu ", (unsigned long) no);

      if (isgroup)
        {
          address_get_personal (address, no, buf, sizeof (buf), &got);

          printf ("group <%s>\n", buf);
        }
      else
        {
          address_get_email (address, no, buf, sizeof (buf), 0);
```

```
            printf ("email <%s>\n", buf);
          }

        address_get_personal (address, no, buf, sizeof (buf), &got);

        if (got && !isgroup)
          printf ("   personal <%s>\n", buf);

        address_get_comments (address, no, buf, sizeof (buf), &got);

        if (got)
          printf ("   comments <%s>\n", buf);

        address_get_local_part (address, no, buf, sizeof (buf), &got);

        if (got)
          {
            printf ("   local-part <%s>", buf);

            address_get_domain (address, no, buf, sizeof (buf), &got);

            if (got)
              printf (" domain <%s>", buf);

            printf ("\n");
          }

        address_get_route (address, no, buf, sizeof (buf), &got);

        if (got)
          printf ("   route <%s>\n", buf);
      }
    address_destroy (&address);

    printf ("\n");

    return 0;
}

static int
parseinput (void)
{
    char buf[BUFSIZ];

    while (fgets (buf, sizeof (buf), stdin) != 0)
      {
        buf[strlen (buf) - 1] = 0;
        parse (buf);
      }
```

```
  return 0;
}

int
main (int argc, const char *argv[])
{
  argc = 1;

  if (!argv[argc])
    {
      return parseinput ();
    }
  for (; argv[argc]; argc++)
    {
      if (strcmp (argv[argc], "-") == 0)
        {
          parseinput ();
        }
      else
        {
          parse (argv[argc]);
        }
    }

  return 0;
}
```

2.14 Locker

```
/* Prefix locker_ is reserved */
#include <mailutils/locker.h>
```

`int` **locker_create** (`locker_t *` *plocker*, `char *`*filename*, `size_t` *len*, `int` *flags*) [Function]

`void` **locker_destroy** (`locker_t *` *plocker*) [Function]

`int` **locker_lock** (`locker_t` *locker*, `int` *flag*) [Function]

```
MU_LOCKER_RDLOCK
MU_LOCKER_WRLOCK
MU_LOCKER_PID
MU_LOCKER_FCNTL
MU_LOCKER_TIME
```

`int` **locker_touchlock** (`locker_t` *locker*) [Function]

`int` **locker_unlock** (`locker_t` *locker*) [Function]

2.15 URL

A mailbox or a mailer can be described in a URL, the string will contain the necessary information to initialize `mailbox_t`, or `mailer_t` properly.

2.15.1 POP3

The POP URL scheme contains a POP server, optional port number and the authentication mechanism. The general form is

```
pop://[<user>[;AUTH=<auth>]@]<host>[:<port>]
 or
pop://[<user>[:<passwd>]@]<host>[:<port>]
```

If *:port* is omitted the default value is 110. Different forms of authentication can be specified with *;AUTH=type*. The special string *;AUTH=** indicates that the client will use a default scheme base on the capability of the server.

```
pop://obelix@gaulois.org
pop://asterix;AUTH=*@france.com
pop://falbala;AUTH=+APOP@france.com
pop://obelix;AUTH=+APOP@village.gaulois.org:2000
pop://obelix:menhir@village.gaulois.org:2000
```

For more complete information see *rfc2368*.

2.15.2 IMAP

The IMAP URL scheme contains an IMAP server, optional port number and the authentication mechanism. The general form is

```
imap://[<user>[;AUTH=<type>]]@<host>[:port][/<mailbox>]
 or
imap://[<user>[:<passwd>]]@<host>[:port][/<mailbox>]
```

If *:port* is omitted the default value is 220. Different forms of authentication can be specified with *;AUTH=type*. The special string *;AUTH=** indicates that the client will use a default scheme base on the capability of the server.

```
imap://obelix@imap.gaulois.org
imap://asterix;AUTH=*@imap.france.com
imap://asterix:potion@imap.france.com
```

For more complete information see *rfc2192*.

2.15.3 File

Local folder should be handle by this URL. It is preferable to let the mailbox recognize the type of mailbox and take the appropriate action.

```
file://path
file://var/mail/user
file://home/obelix/Mail
```

For MMDF, MH local mailboxes URLs are provided, but it is preferable to use `file://path` and let the library figure out which one.

```
mmdf://path
mh://path
```

2.15.4 Mailto

After setting a mailer, `mailto:` is used to tell the mailer where and to whom the message is for.

```
mailto://hostname
```

Mailto can be used to generate short messages, for example to subscribe to mailing lists.

```
mailto://bug-mailutils@gnu.org?body=subscribe
mailto://bug-mailutils@gnu.org?Subject=hello&body=subscribe
```

For more complete information see *rfc2368*.

2.15.5 URL functions

Helper functions are provided to retrieve and set the *URL* fields.

`int` **url_create** (`url_t *`*url*, `const char *`*name*) [Function]
Create and the *url* data structure, but do not parse it.

`void` **url_destroy** (`url_t *`) [Function]
Destroy the url and free it's resources.

`int` **url_parse** (`url_t` *url*) [Function]
Parses the url, after calling this the get functions can be called.

The syntax, condensed from RFC 1738, and extended with the ;auth= of RFC 2384 (for POP) and RFC 2192 (for IMAP) is:

```
url =
    scheme ":" [ "//"

    [ user [ ( ":" password ) | ( ";auth=" auth ) ] "¨ ]

    host [ ":" port ]

    [ ( "/" urlpath ) | ( "?" query ) ] ]
```

This is a generalized URL syntax, and may not be exactly appropriate for any particular scheme.

`const` **char*** `url_to_string` (`const url_t` *url*) [Function]

`int` **url_get_scheme** (`const url_t` *url*, `char *`*schem*, `size_t` *len*, `size_t *`*n*) [Function]

`int` **url_get_user** (`const url_t` *url*, `char *`*usr*, `size_t` *len*, `size_t *`*n*) [Function]

`int` **url_get_passwd** (`const url_t` *url*, `char *`*passwd*, `size_t` *len*, `size_t *`*n*) [Function]

`int` **url_get_host** (`const url_t` *url*, `char *`*host*, `size_t` *len*, `size_t *`*n*) [Function]

`int` **url_get_port** (`const url_t` *url*, `long *`*port*) [Function]

`int` **url_get_path** (`const url_t` *url*, `char *`*path*, `size_t` *len*, `size_t *`*n*) [Function]

`int` **url_get_query** (`const url_t` *url*, `char *`*query*, `size_tlen`, `size_t *`*n*) [Function]

`char*` **url_decode** (`const char*` *string*) [Function]

Decodes an RFC 1738 % encoded string, returning the decoded string in allocated memory. If the string is not encoded, this degenerates to a strdup().

2.16 Example

```
#include <stdio.h>
#include <string.h>
#include <mailutils/errno.h>
#include <mailutils/url.h>

int
main ()
{
  char str[1024];
  char buffer[1024];
  long port = 0;
  int len = sizeof (buffer);
  url_t u = NULL;

  while (fgets (str, sizeof (str), stdin) != NULL)
    {
      int rc;

      str[strlen (str) - 1] = '\0';     /* chop newline */
      if (strspn (str, " \t") == strlen (str))
        continue;               /* skip empty lines */
      if ((rc = url_create (&u, str)) != 0)
        {
          fprintf (stderr, "url_create %s ERROR: [%d] %s",
                   str, rc, mu_errstring (rc));
          exit (1);
        }
      if ((rc = url_parse (u)) != 0)
        {
          printf ("%s => FAILED: [%d] %s\n",
                  str, rc, mu_errstring (rc));
          continue;
        }
      printf ("%s => SUCCESS\n", str);
```

```
        url_get_scheme (u, buffer, len, NULL);
        printf ("\tscheme <%s>\n", buffer);

        url_get_user (u, buffer, len, NULL);
        printf ("\tuser <%s>\n", buffer);

        url_get_passwd (u, buffer, len, NULL);
        printf ("\tpasswd <%s>\n", buffer);

        url_get_auth (u, buffer, len, NULL);
        printf ("\tauth <%s>\n", buffer);

        url_get_host (u, buffer, len, NULL);
        printf ("\thost <%s>\n", buffer);

        url_get_port (u, &port);
        printf ("\tport %ld\n", port);

        url_get_path (u, buffer, len, NULL);
        printf ("\tpath <%s>\n", buffer);

        url_get_query (u, buffer, len, NULL);
        printf ("\tquery <%s>\n", buffer);

        url_destroy (&u);

      }
    return 0;
  }
```

2.17 Parse822 — Functions for Parsing RFC 822 Headers

```
/* Prefix parse822_ is reserved */
#include <mailutils/parse822.h>
```

Internet Message Format, see Address node for the discussion.

`int` **parse822_address_list** (`address_t* a, const char* s`) [Function]

`int` **parse822_mail_box** (`const char** p, const char* e, address_t* a`) [Function]

`int` **parse822_group** (`const char** p, const char* e, address_t* a`) [Function]

`int` **parse822_address** (`const char** p, const char* e, address_t* a`) [Function]

`int` **parse822_route_addr** (`const char** p, const char* e, address_t* a`) [Function]

`int` **parse822_route** (`const char** p, const char* e, char** route`) [Function]

`int` **parse822_addr_spec** (`const char** p, const char* e, address_t* a`) [Function]

`int` **parse822_unix_mbox** (`const char** p, const char* e, address_t* a`) [Function]

`int` **parse822_local_part** (`const char** p, const char* e, char** local_part`) [Function]

`int` **parse822_domain** (`const char** p, const char* e, char** domain`) [Function]

`int` **parse822_sub_domain** (`const char** p, const char* e, char** sub_domain`) [Function]

`int` **parse822_domain_ref** (`const char** p, const char* e, char** domain_ref`) [Function]

`int` **parse822_domain_literal** (`const char** p, const char* e, char** domain_literal`) [Function]

`int` **parse822_quote_string** (`char** quoted, const char* raw`) [Function]

`int` **parse822_quote_local_part** (`char** quoted, const char* raw`) [Function]

`int` **parse822_field_body** (`const char** p, const char *e, char** fieldbody`) [Function]

`int` **parse822_field_name** (`const char** p, const char *e, char** fieldname`) [Function]

3 Authentication Library

The functions from '`libmailbox`' library get user information from the system user database. The library '`libmuauth`' extends this functionality, allowing '`libmailbox`' functions to obtain information about a user from several places, like SQL database, etc. The method used is described in detail in Section 6.1 [authentication], page 72. This chapter contains a very succinct description of the underlying library mechanism.

3.1 Data Types

mu_auth_fp [Data Type]

This is a pointer to authentication or authorization data. It is defined as follows:

```
typedef int (*mu_auth_fp) (void *return_data,
                           void *key,
                           void *func_data,
                           void *call_data);
```

Its arguments are:

return_data

[1]. Upon successful return authorization handler leaves in this memory location a pointer to the filled `mu_auth_data` structure with the user's information.

For authentication handlers this argument is always `NULL` and should be ignored.

key The search key value. Its actual type depends upon type of the handler.

For authorization handlers it is `const char*` if the handler is called by `mu_get_auth_by_name()` and `uid_t *` if it is called by `mu_get_auth_by_uid()`.

For authentication handlers it is always `struct mu_auth_data*` representing the user's data obtained by a previous call to a `mu_get_auth_by_...` function.

func_data Any data associated with this handler.

call_data Any call specific data. This argument is not used at the moment.

mu_auth_data [Data Type]

The `mu_auth_data` is used to return the information about the user. It is similar to system `struct passwd`, except that it is more mailutils-specific. Its definition is:

```
struct mu_auth_data {
  /* These are from struct passwd */
  char    *name;       /* user name */
  char    *passwd;     /* user password */
  uid_t   uid;         /* user id */
  gid_t   gid;         /* group id */
  char    *gecos;      /* real name */
  char    *dir;        /* home directory */
  char    *shell;      /* shell program */
  /* */
  char    *mailbox;    /* Path to the user's system mailbox */
  int     change_uid;  /* Should the uid be changed? */
};
```

[1] Actually it shoud have been `struct mu_auth_data** return_data`. This will be fixed in the next release

mu_auth_module [Data Type]

The `mu_auth_module` structure contains full information about a libmuauth module. It is declared as follows:

```
struct mu_auth_module {
  char           *name;              /* Module name */
  struct argp    *argp;              /* Corresponding argp structure */
  mu_auth_fp     authenticate;       /* Authentication function ... */
  void           *authenticate_data; /* ... and its specific data */
  mu_auth_fp     auth_by_name;       /* Get user info by user name */
  void           *auth_by_name_data; /* ... and its specific data */
  mu_auth_fp     auth_by_uid;        /* Get user info by user id */
  void           *auth_by_uid_data;  /* ... and its specific data */
};
```

3.2 Initializing the libmuauth

`void` **mu_auth_init** (`void`) [Function]

This function registers the command line capability "auth". It must be called after registering '`libmuauth`' modules and before calling `mu_agrp_parse()`. If an error occurs, this function prints diagnostic message and aborts the program.

`void` **MU_AUTH_REGISTER_ALL_MODULES** (`void`) [Function]

This macro registers all default modules and calls `mu_auth_init()`.

3.3 Module Creation and Destruction

`int` **mu_auth_data_alloc** (`struct mu_auth_data **ptr, const char *name, const char *passwd, uid_t uid, gid_t gid, const char *gecos, const char *dir, const char *shell, const char *mailbox, int change_uid`) [Function]

Create a `mu_auth_data` structure and initialize it with the given values. Returns 0 on success and 1 otherwise.

`void` **mu_auth_data_free** (`struct mu_auth_data *`*ptr*) [Function]

Free the `mu_auth_data` structure allocated by a call to `mu_auth_data_alloc()`.

`void` **mu_auth_register_module** (`struct mu_auth_module *`*mod*) [Function]

Register the module defined by the *mod* argument.

3.4 Obtaining Authorization Information

`int` **mu_auth_runlist** (`list_t` *flist*, `void *`*return_data*, `void *`*key*, `void *`*call_data*); [Function]

The list is expected to contain `mu_auth_fp` pointers. Each of them is dereferenced and executed until either the list is exhausted or any of the functions returns non-zero, whichever occurs first. The *return_data* and *key* arguments are passed as the first two parameters to the function (see the definition of `mu_auth_fp`, notice the footnote), the `call_data` is passed as its last parameter.

The function returns 0 if none of the functions from `list` succeeded, i.e. returned non-zero value. Otherwise it returns the return code from the succeeded function.

`struct mu_auth_data *` **mu_get_auth_by_name** (`const char *`*username*) [Function]

Search the information about given user by its username. Similar to system's `getpwnam` call).

`struct mu_auth_data *` **mu_get_auth_by_uid** (`uid_t` *uid*) [Function]

Search the information about given user by its uid. Similar to system's `getpwuid` call).

`int` **mu_authenticate** (`struct mu_auth_data *`*auth_data*, `char *`*pass*) [Function]

Authenticate the user whose data are in *auth_data* using password *pass*. Return 0 if the user is authenticated.

3.5 Existing Modules

`int` **mu_auth_nosupport** (`void *return_data, void *key, void *func_data, void *call_data`); [Function]

The "not-supported" module. Always returns `ENOSYS`.

mu_auth_system_module [Variable]

This module is always registered even if '`libmuauth`' is not linked. It performs usual authentication using system user database ('`/etc/password`' et al.)

mu_auth_generic_module [Variable]

This module is always registered even if '`libmuauth`' is not linked. Both its authorization handlers are `mu_auth_nosupport`. Its authentication handler computes the MD5 or DES hash over the supplied password with the seed taken from `passwd` member of its *key* argument. Then it compares the obtained hash with the `passwd` member itself and returns 1 if both strings match.

mu_auth_pam_module [Variable]

Implements PAM authentication. Both authorization handlers are `mu_auth_nosupport()`.

mu_auth_sql_module [Variable]

Implements authentication and authorization via MySQL database. The credentials for accessing the database are taken from global variables `sql_host`, `sql_port`, `sql_user`, `sql_passwd` and `sql_db`. The SQL queries for retrieving user information from global variables `sql_getpwnam_query` and `sql_getpwuid_query`. The variable `sql_getpass_query` keeps the query used for retrieving user's password. See Section 6.2.6 [auth], page 76, for information on command line options used to set these variables.

mu_auth_virtual_module [Variable]

Implements `mu_get_auth_by_name` method using virtual mail domains. Neither `mu_get_auth_by_uid` nor `mu_authenticate` is implemented. This module must be used together with `generic` module.

3.6 Using Libmuauth in Your Programs

To link your program against '`libmuauth`', obtain loader arguments by running `mailutils-config` as follows:

```
mailutils-config --link auth
```

See Section 6.13 [mailutils-config], page 118, for more information about this utility.

Here is a sample Makefile fragment:

```
MU_LDFLAGS=`mailutils-config --link auth`
MU_INCLUDES=`mailutils-config --include`

myprog: myprog.c
        $(CC) -omyprog $(CFLAGS) $(MU_INCLUDES) myprog.c $(MU_LDFLAGS)
```

If your program will be using only default modules provided by the library, then it will suffice to call `MU_AUTH_REGISTER_ALL_MODULES()` somewhere near the start of your program. As an example, consider the following code fragment (it is taken from the `imap4d` daemon):

```
int
main (int argc, char **argv)
{
  struct group *gr;
  int status = EXIT_SUCCESS;

  state = STATE_NONAUTH; /* Starting state in non-auth.  */

  MU_AUTH_REGISTER_ALL_MODULES ();
  mu_argp_parse (&argp, &argc, &argv, 0, imap4d_capa,
                 NULL, &daemon_param);
  ...
```

Otherwise, if your program will use it's own modules, first register them with `mu_auth_register_module` and then call `mu_auth_init()`, e.g.:

```
struct mu_auth_module radius_module = {
  ...
};

struct mu_auth_module ext_module = {
  ...
};

int
main (int argc, char **argv)
{
  mu_auth_register_module (&radius_module);
  mu_auth_register_module (&ext_module);
  mu_auth_init ();

  ...
```

These two approaches may be combined, allowing you to use both your modules and the ones provided by Mailutils. Consider the example below:

```
int
main (int argc, char **argv)
{
  mu_auth_register_module (&radius_module);
  mu_auth_register_module (&ext_module);
  MU_AUTH_REGISTER_ALL_MODULES ();

  ...
}
```

4 Mailutils to Scheme interface

The library '`libmu_scm`' provides an interface between Mailutils and Guile, allowing to access the Mailutils functionality from a Scheme program. For more information about Guile, refer to section "Overview" in *The Guile Reference Manual*. For information about Scheme programming language, See section "Top" in *Revised(4) Report on the Algorithmic Language Scheme*.

4.1 Address Functions

mu-address-get-personal *ADDRESS NUM* [Scheme Function]
Return personal part of an email address.

mu-address-get-comments *ADDRESS NUM* [Scheme Function]

mu-address-get-email *ADDRESS NUM* [Scheme Function]
Return email part of an email address.

mu-address-get-domain *ADDRESS NUM* [Scheme Function]
Return domain part of an email address

mu-address-get-local *ADDRESS NUM* [Scheme Function]
Return local part of an email address.

mu-address-get-count *ADDRESS* [Scheme Function]
Return number of parts in email address.

4.2 Mailbox Functions

mu-mailbox-open *URL MODE* [Scheme Function]
Opens a mailbox specified by URL.

mu-mailbox-close *MBOX* [Scheme Function]
Closes mailbox MBOX

mu-mailbox-get-url *MBOX* [Scheme Function]
Returns the URL of the mailbox.

mu-mailbox-get-port *MBOX MODE* [Scheme Function]
Returns a port associated with the contents of the MBOX. MODE is a string defining operation mode of the stream. It may contain any of the two characters: '`r`' for reading, '`w`' for writing.

mu-mailbox-get-message *MBOX MSGNO* [Scheme Function]
Retrieve from MBOX message # MSGNO.

mu-mailbox-messages-count *MBOX* [Scheme Function]
Returns number of messages in the mailbox.

mu-mailbox-expunge *MBOX* [Scheme Function]
Expunges deleted messages from the mailbox.

mu-mailbox-url *MBOX* [Scheme Function]
Returns the URL of the mailbox

mu-mailbox-append-message *MBOX MESG* [Scheme Function]
Appends the message to the mailbox

4.3 Message Functions

mu-message-copy *MESG* [Scheme Function]
Creates the copy of the given message.

mu-message-set-header *MESG HEADER VALUE REPLACE* [Scheme Function]
Sets new VALUE to the header HEADER of the message MESG. If the HEADER is already present in the message its value is replaced with the supplied one if the optional REPLACE is #t. Otherwise new header is created and appended.

mu-message-get-size *MESG* [Scheme Function]
Returns the size of the given message.

mu-message-get-lines *MESG* [Scheme Function]
Returns number of lines in the given message.

mu-message-get-sender *MESG* [Scheme Function]
Returns the sender email address for the message MESG.

mu-message-get-header *MESG HEADER* [Scheme Function]
Returns the header value of the HEADER in the MESG.

mu-message-get-header-fields *MESG HEADERS* [Scheme Function]
Returns the list of headers in the MESG. If optional HEADERS is specified it should be a list of header names to restrict return value to.

mu-message-set-header-fields *MESG LIST REPLACE* [Scheme Function]
Set the headers in the message MESG from LIST LIST is a list of (cons HEADER VALUE) Optional parameter REPLACE specifies whether the new header values should replace the headers already present in the message.

mu-message-delete *MESG FLAG* [Scheme Function]
Mark given message as deleted. Optional FLAG allows to toggle deleted mark The message is deleted if it is #t and undeleted if it is #f

mu-message-get-flag *MESG FLAG* [Scheme Function]
Return value of the attribute FLAG.

mu-message-set-flag *MESG FLAG VALUE* [Scheme Function]
Set the given attribute of the message. If optional VALUE is #f, the attribute is unset.

mu-message-get-user-flag *MESG FLAG* [Scheme Function]
Returns value of the user attribute FLAG.

mu-message-set-user-flag *MESG FLAG VALUE* [Scheme Function]
Set the given user attribute of the message. If optional VALUE is #f, the attribute is unset.

mu-message-get-port *MESG MODE FULL* [Scheme Function]
Returns a port associated with the given MESG. MODE is a string defining operation mode of the stream. It may contain any of the two characters: '`r`' for reading, '`w`' for writing. If optional FULL argument specified, it should be a boolean value. If it is #t then the returned port will allow access to any part of the message (including headers). If it is #f then the port accesses only the message body (the default).

mu-message-get-body *MESG* [Scheme Function]
Returns the message body for the message MESG.

mu-message-send *MESG MAILER* [Scheme Function]
Sends the message MESG. Optional MAILER overrides default mailer settings in mu-mailer.

4.4 MIME Functions

mu-mime-create *FLAGS MESG* [Scheme Function]
Creates a new MIME object.

mu-mime-multipart? *MIME* [Scheme Function]
Returns #t if MIME is a multipart object.

mu-mime-get-num-parts *MIME* [Scheme Function]
Returns number of parts in a MIME object.

mu-mime-get-part *MIME PART* [Scheme Function]
Returns part number PART from a MIME object.

mu-mime-add-part *MIME MESG* [Scheme Function]
Adds MESG to the MIME object.

mu-mime-get-message *MIME* [Scheme Function]
Converts MIME object to a message.

4.5 Log Functions

mu-openlog *IDENT OPTION FACILITY* [Scheme Function]
Opens a connection to the system logger for Guile program.

mu-logger *PRIO TEXT* [Scheme Function]
Generates a log message to be distributed via syslogd.

mu-closelog [Scheme Function]
Closes the channel to the system logger open by mu-openlog.

4.6 Direct Linking

If you plan to link your program directly to 'libguile', it will probably make sense to link 'libmu_scm' directly as well. The arguments to the program loader may be obtained by running

```
mailutils-config --link guile
```

See Section 6.13 [mailutils-config], page 118, for more information about this utility.

Here is a sample Makefile fragment:

```
MU_LDFLAGS=`mailutils-config --link guile`
MU_INCLUDES=`mailutils-config --include`

myprog: myprog.c
        $(CC) -omyprog $(CFLAGS) $(MU_INCLUDES) myprog.c $(MU_LDFLAGS)
```

4.7 Dynamic Linking

Dynamic linking is the preferred method of using 'libmu_scm'. It uses Guile "use-modules" mechanism. An interface module 'mailutils.scm' is provided in order to facilitate using this method. This module is installed in the package data directory (by default it is *prefix*/share/mailutils). A sample use of this module is:

```
(set! %load-path (list "/usr/local/share/mailutils"))
(use-modules (mailutils))

# Now you may use mailutils functions:

(let ((mb (mu-mailbox-open "/var/spool/mail/gray" "r")))
...
```

Note, that you should explicitly modify the `%load-path` before calling `use-modules`, otherwise Guile will not be able to find 'mailutils.scm'.

5 Sieve Library

This chapter describes GNU Sieve library.

5.1 Library Description

`Libsieve` is GNU implementation of the mail filtering language Sieve.

The library is built around a *Sieve Machine* — an abstract computer constructed specially to handle mail filtering tasks. This computer has two registers: program counter and numeric accumulator; a runtime stack of unlimited depth and the code segment. A set of functions is provided for creating and destroying instances of Sieve Machine, manipulating its internal data, compiling and executing a sieve program.

The following is a typical scenario of using `libsieve`:

1. Application program creates the instance of sieve machine.

2. Then `sieve_compile` function is called to translate the Sieve source into an equivalent program executable by the Machine
3. A mailbox is opened and associated with the Machine
4. The Machine executes the program over the mailbox
5. When the execution of the program is finished, all messages upon which an action was executed other than `keep` are marked with the delete flag. Thus, running `mailbox_expunge` upon the mailbox finishes the job, leaving in the mailbox only those messages that were preserved by the filter.
6. Finally, the instance of Sieve Machine is destroyed and the resources allocated for it are reclaimed.

The following sections describe in detail the functions from the Sieve Library.

5.1.1 Sieve Data Types

sieve_machine_t [Data Type]

This is an opaque data type representing a pointer to an instance of sieve machine. The sieve_machine_t keeps all information necessary for compiling and executing the script.

It is created by `sieve_machine_create()` and destroyed by `sieve_machine_destroy()`. The functions for manipulating this data type are described in Section 5.1.2 [Manipulating the Sieve Machine], page 49.

sieve_data_type [Data Type]

This enumeration keeps the possible types of sieve data. These are:

`SVT_VOID` No datatype.

`SVT_NUMBER`
: Numeric type.

`SVT_STRING`
: Character string.

`SVT_STRING_LIST`
: A `list_t`. Each item in this list represents a character string.

`SVT_TAG` A sieve tag. See `sieve_runtime_tag_t` below.

`SVT_IDENT`
: A character string representing an identifier.

`SVT_VALUE_LIST`
: A `list_t`. Each item in this list is of `sieve_value_t`.

`SVT_POINTER`
: An opaque pointer.

sieve_value_t [Data Type]

The `sieve_value_t` keeps an instance of sieve data. It is defined as follows:

```
typedef struct {
  sieve_data_type type;        /* Type of the data */
  union {
    char *string;              /* String value or identifier */
    size_t number;               /* Numeric value */
    list_t list;               /* List value */
    sieve_runtime_tag_t *tag;  /* Tag value */
    void *ptr;                 /* Pointer value */
  } v;
} sieve_value_t;
```

Depending on the value of `type` member, following members of the union `v` keep the actual value:

`SVT_VOID` Never appears.

`SVT_NUMBER`
: The numeric value is kept in `number` member.

`SVT_STRING`
: The string is kept in `string` member.

`SVT_STRING_LIST`
`SVT_VALUE_LIST`
: The list itself is pointed to by `list` member

`SVT_TAG` The tag value is pointed to by `tag` member.

`SVT_IDENT`
: The `string` member points to the identifier name.

`SVT_POINTER`
: The data are pointed to by `ptr` member.

sieve_tag_def_t [Data Type]

This structure represents a definition of a tagged (optional) argument to a sieve action or test. It is defined as follows:

```
typedef struct {
  char *name;               /* Tag name */
  sieve_data_type argtype; /* Type of tag argument. */
} sieve_tag_def_t;
```

The `name` member points to the tag's name *without leading colon*. The `argtype` is set to `SVT_VOID` if the tag does not take argument, or to the type of the argument otherwise.

sieve_runtime_tag_t [Data Type]

This structure represents the tagged (optional) argument at a runtime. It is defined as:

```
struct sieve_runtime_tag {
  char *tag;                /* Tag name */
  sieve_value_t *arg;       /* Tag argument (if any) */
};
```

The `arg` member is `NULL` if the tag does not take an argument.

sieve_handler_t [Data Type]

This is a pointer to function handler for a sieve action or test. It is defined as follows:

```
typedef int (*sieve_handler_t) (sieve_machine_t mach,
                                list_t args, list_t tags);
```

The arguments to the handler have the following meaning:

mach Sieve machine being processed.

args A list of required arguments to the handler

tags A list of optional arguments (tags).

sieve_printf_t [Data Type]

A pointer to a diagnostic output function. It is defined as follows:

```
typedef int (*sieve_printf_t) (void *data, const char *fmt, va_list ap);
```

data A pointer to application specific data. These data are passed as second argument to `sieve_machine_init()`.

fmt Printf-like format string.

ap Other arguments.

sieve_parse_error_t [Data Type]

This data type is declared as follows:

```
typedef int (*sieve_parse_error_t) (void *data,
                                    const char *filename, int lineno,
                                    const char *fmt, va_list ap);
```

It is used to declare error handlers for parsing errors. The application-specific data are passed in the *data* argument. Arguments *filename* and *line* indicate the location of the error in the source text, while *fmt* and *ap* give verbose description of the error.

sieve_action_log_t [Data Type]

A pointer to the application-specific logging function:

```
typedef void (*sieve_action_log_t) (void *data,
                                    const char *script,
                                    size_t msgno, message_t msg,
                                    const char *action,
                                    const char *fmt, va_list ap);
```

data Application-specific data.

script Name of the sieve script being executed.

msgno Ordinal number of the message in mailbox, if appropriate. When execution is started using `sieve_message()`, this argument is zero.

msg The message this action is executed upon.

action The name of the action.

fmt
var These two arguments give the detailed description of the action.

sieve_comparator_t [Data Type]

```
typedef int (*sieve_comparator_t) (const char *, const char *);
```

A pointer to the comparator handler function. The function compares its two operands and returns 1 if they are equal, and 0 otherwise. *Notice*, that the sense of the return value is inverted in comparison with most standard libc functions like `stcmp()`, etc.

sieve_retrieve_t [Data Type]

```
typedef int (*sieve_retrieve_t) (void *item, void *data, int idx,
                                 char **pval);
```

A pointer to generic retriever function. See description of `sieve_vlist_compare()` for details of its usage.

sieve_destructor_t [Data Type]

```
typedef void (*sieve_destructor_t) (void *data);
```

A pointer to destructor function. The function frees any resources associated with `data`. See the description of `sieve_machine_add_destructor()` for more information.

sieve_tag_checker_t [Data Type]

```
typedef int (*sieve_tag_checker_t) (const char *name,
                                    list_t tags,
                                    list_t args)
```

A pointer to tag checker function. The purpose of the function is to perform compilation-time consistency test on tags. Its arguments are:

name Name of the test or action whose tags are being checked.

tags A list of `sieve_runtime_tag_t` representing tags.

args A list of `sieve_value_t` representing required arguments to *name*.

The function is allowed to make any changes in *tags* and *args*. It should return 0 if the syntax is correct and non-zero otherwise. It is responsible for issuing the diagnostics in the latter case. [FIXME: describe how to do that]

5.1.2 Manipulating the Sieve Machine

This subsection describes functions used to create an instance of the sieve machine, read or alter its internal fields and destroy it.

`int` **sieve_machine_init** (`sieve_machine_t *`*mach*`, void *`*data*) [Function]

The `sieve_machine_init()` function creates an instance of a sieve machine. A pointer to the instance itself is returned in the argument *mach*. The user-specific data to be associated with the new machine are passed in *data* argument. The function returns 0 on success, non-zero error code otherwise,

`void` **sieve_machine_destroy** (`sieve_machine_t *`*pmach*) [Function]

This function destroys the instance of sieve machine pointed to by *mach* parameter. After execution of `sieve_machine_destroy()` *pmach* contains `NULL`. The destructors registered with `sieve_machine_add_destructor()` are executed in LIFO order.

`int` **sieve_machine_add_destructor** (`sieve_machine_t` *mach*, `sieve_destructor_t` *destr*, `void *`*ptr*); [Function]

This function registers a destructor function *dest*. The purpose of the destructor is to free any resources associated with the item *ptr*. The destructor function takes a single argument — a pointer to the data being destroyed. All registered destructors are called in reverse order upon execution of `sieve_machine_destroy()`. Here's a short example of the use of this function:

```
static void
free_regex (void *data)
{
  regfree ((regex_t*)data);
}

int
match_part_checker (const char *name, list_t tags, list_t args)
{
  regex_t *regex;

  /* Initialise the regex: */
  regex = sieve_malloc (mach, sizeof (*regex));
  /* Make sure it will be freed when necessary */
  sieve_machine_add_destructor (sieve_machine, free_regex, regex);
  .
  .
  .
}
```

`void` ***sieve_get_data** (`sieve_machine_t` *mach*) [Function]

This function returns the application-specific data associated with the instance of sieve machine. See `sieve_machine_init()`.

`message_t` **sieve_get_message** (`sieve_machine_t` *mach*) [Function]

This function returns the current message.

`size_t` **sieve_get_message_num** (`sieve_machine_t` *mach*); [Function]

This function returns the current message number in the mailbox. If there are no mailbox, i.e. the execution of the sieve code is started with `sieve_message`, this function returns 1.

`int` **sieve_get_debug_level** (`sieve_machine_t` *mach*) [Function]

Returns the debug level set for this instance of sieve machine.

`ticket_t` **sieve_get_ticket** (`sieve_machine_t` *mach*) [Function]

Returns the authentication ticket for this machine.

`mailer_t` **sieve_get_mailer** (`sieve_machine_t` *mach*) [Function]

Returns the mailer.

`char *` **sieve_get_daemon_email** `__P((sieve_machine_t` *mach*) [Function]

This function returns the *daemon email* associated with this instance of sieve machine. The daemon email is an email address used in envelope from addresses of automatic reply messages. By default its local part is '`<MAILER-DAEMON>`' and the domain part is the machine name.

`void` **sieve_set_error** (`sieve_machine_t` *mach*, `sieve_printf_t` [Function]
error_printer)

This function sets the error printer function for the machine. If it is not set, the default error printer will be used. It is defined as follows:

```
int
_sieve_default_error_printer (void *unused, const char *fmt,
                              va_list ap)
{
  return mu_verror (fmt, ap);
}
```

`void` **sieve_set_parse_error** (`sieve_machine_t` *mach*, [Function]
`sieve_parse_error_t` *p*)

This function sets the parse error printer function for the machine. If it is not set, the default parse error printer will be used. It is defined as follows:

```
int
_sieve_default_parse_error (void *unused,
                            const char *filename, int lineno,
    const char *fmt, va_list ap)
{
  if (filename)
    fprintf (stderr, "%s:%d: ", filename, lineno);
  vfprintf (stderr, fmt, ap);
  fprintf (stderr, "\n");
  return 0;
}
```

`void` **sieve_set_debug** (`sieve_machine_t` *mach*, `sieve_printf_t` [Function]
debug));

This function sets the debug printer function for the machine. If it is not set, the default debug printer is `NULL` which means no debugging information will be displayed.

`void` **sieve_set_debug_level** (`sieve_machine_t` *mach*, `mu_debug_t` [Function]
dbg, `int` *level*)

This function sets the debug level for the given instance of sieve machine. The *dbg* argument is the `mu_debug_t` object to be used with mailutils library, the *level* argument specifies the debugging level for the sieve library itself. It is a bitwise or of the following values:

`MU_SIEVE_DEBUG_TRACE`
: Trace the execution of the sieve script.

`MU_SIEVE_DEBUG_INSTR`
: Print the sieve machine instructions as they are executed.

`MU_SIEVE_DEBUG_DISAS`
: Dump the disassembled code of the sieve machine. Do not run it.

`MU_SIEVE_DRY_RUN`
: Do not executed the actions, only show what would have been done.

`void` **sieve_set_logger** (`sieve_machine_t` *mach*, `sieve_action_log_t` *logger*) [Function]
This function sets the logger function. By default the logger function is `NULL`, which means that the executed actions are not logged.

`void` **sieve_set_ticket** (`sieve_machine_t` *mach*, `ticket_t` *ticket*) [Function]
This function sets the authentication ticket to be used with this machine.

`void` **sieve_set_mailer** (`sieve_machine_t` *mach*, `mailer_t` *mailer*) [Function]
This function sets the mailer. The default mailer is `"sendmail:"`.

`void` **sieve_set_daemon_email** (`sieve_machine_t` *mach*, `const char` **email*) [Function]
This functions sets the *daemon email* for `reject` and `redirect` actions.

`int` **sieve_is_dry_run** (`sieve_machine_t` *mach*) [Function]
The `sieve_is_dry_run()` returns 1 if the machine is in *dry run* state, i.e. it will only log the actions that would have been executed without actually executing them. The dry run state is set by calling `sieve_set_debug_level()` if its last argument has the `MU_SIEVE_DRY_RUN` bit set.

`const char *` **sieve_type_str** (`sieve_data_type` *type*) [Function]
Returns the string representation for the given sieve data type. The return value is a pointer to a static constant string.

5.1.3 Logging and Diagnostic Functions

`void` **sieve_error** (`sieve_machine_t` *mach*, `const char *`*fmt*, ...) [Function]
Format and output an error message using error printer of the machine *mach*.

`void` **sieve_debug** (`sieve_machine_t` *mach*, `const char *`*fmt*, ...) [Function]
Format and output a debug message using debug printer of the machine *mach*.

`void` **sieve_log_action** (`sieve_machine_t` *mach*, `const char *`*action*, `const char *`*fmt*, ...) [Function]
Log a sieve action using logger function associated with the machine *mach*.

`void` **sieve_abort** (`sieve_machine_t` *mach*) [Function]
Immediately abort the execution of the script.

5.1.4 Symbol Space Functions

`sieve_register_t *` **sieve_test_lookup** (`sieve_machine_t` *mach*, `const char *`*name*) [Function]

Find a register object describing the test *name*. Returns `NULL` if no such test exists.

`sieve_register_t *` **sieve_action_lookup** (`sieve_machine_t` *mach*, `const char *`*name*) [Function]

Find a register object describing the action *name*. Returns `NULL` if no such action exists.

`int` **sieve_register_test** (`sieve_machine_t` *mach*, `const char *`*name*, `sieve_handler_t` *handler*, `sieve_data_type *`*arg_types*, `sieve_tag_group_t *`*tags*, `int` *required*) [Function]

`int` **sieve_register_action** (`sieve_machine_t` *mach*, `const char *`*name*, `sieve_handler_t` *handler*, `sieve_data_type *`*arg_types*, `sieve_tag_group_t *`*tags*, `int` *required*) [Function]

`int` **sieve_register_comparator** (`sieve_machine_t` *mach*, `const char *`*name*, `int` *required*, `sieve_comparator_t` *is*, `sieve_comparator_t` *contains*, `sieve_comparator_t` *matches*, `sieve_comparator_t` *regex*) [Function]

`int` **sieve_tag_lookup** (`list_t` *taglist*, `char *`*name*, `sieve_value_t **`*arg*) [Function]

`int` **sieve_load_ext** (`sieve_machine_t` *mach*, `const char *`*name*) [Function]

5.1.5 Memory Allocation

The following functions act as their libc counterparts. The allocated memory is associated with the *mach* argument and is automatically freed upon the call to `sieve_machine_destroy (`*mach*`)`.

`void *` **sieve_malloc** (`sieve_machine_t` *mach*, `size_t` *size*) [Function]

Allocates *size* bytes and returns a pointer to the allocated memory.

`char *` **sieve_mstrdup** (`sieve_machine_t` *mach*, `const char *`*str*) [Function]

This function returns a pointer to a new string which is a duplicate of the string *str*.

`void *` **sieve_mrealloc** (`sieve_machine_t` *mach*, `void *`*ptr*, `size_t` *size*) [Function]

Changes the size of the memory block pointed to by *ptr* to *size* bytes. The contents will be unchanged to the minimum of the old and new sizes; newly allocated memory will be uninitialized. If *ptr* is `NULL`, the call is equivalent to `sieve_malloc(`*mach*`, `*size*`)`; if *size* is equal to zero, the call is equivalent to `sieve_mfree(`*ptr*`)`. Unless *ptr* is `NULL`, it must have been returned by an earlier call to `sieve_malloc()` or `sieve_mrealloc()`.

`void` **sieve_mfree** (`sieve_machine_t` *mach*, `void *`*ptr*) [Function]
`sieve_mfree()` frees the memory space pointed to by *ptr* and detaches it from the destructor list of *mach*. The *ptr* must have been returned by a previous call to `sieve_malloc()` or `sieve_mrealloc()`. Otherwise, or if `sieve_mfree(`*ptr*`)` has already been called before, undefined behaviour occurs.

If *ptr* is `NULL`, no operation is performed.

5.1.6 Compiling and Executing the Script

`int` **sieve_compile** (`sieve_machine_t` *mach*, `const char *`*name*) [Function]
Compile the sieve script from the file *name*.

`int` **sieve_mailbox** (`sieve_machine_t` *mach*, `mailbox_t` *mbox*) [Function]
Execute the code from the given instance of sieve machine *mach* over each message in the mailbox *mbox*.

`int` **sieve_message** (`sieve_machine_t` *mach*, `message_t` *message*) [Function]
Execute the code from the given instance of sieve machine *mach* over the *message*.

`int` **sieve_disass** (`sieve_machine_t` *mach*) [Function]
Dump the disassembled code of the sieve machine *mach*.

5.2 Input Language

The input language understood by the GNU Sieve Library is a superset of the Sieve language as described in RFC 3028.

5.2.1 Lexical Structure

Whitespace and Comments

Comments are semantically equivalent to whitespace and can be used anyplace that whitespace is (with one exception in multi-line strings, as described below).

There are two kinds of comments: hash comments, that begin with a '`#`' character that is not contained within a string and continue until the next newline, and C-style or bracketed comments, that are delimited by '`/*`' and '`*/`' tokens. The bracketed comments may span multiple lines. E.g.:

```
if size :over 100K
  { # this is a comment
    discard;
  }

if size :over 100K
  { /* this is a comment
       this is still a comment */ discard /* this is a comment again
```

```
      */ ;
}
```

Like in C, bracketed comments do not nest.

Lexical Tokens

The basic lexical entities are *identifiers* and *literals*.

An *identifier* is a sequence of letters, digits and underscores, started with a letter or underscore. For example, `header` and `check_822_again` are valid identifiers, whereas `1st` is not. A special form of identifier is *tag*: it is an identifier prefixed with a colon (':'), e.g.: `:comparator`.

A *literal* is a data that is not executed, merely evaluated "as is", to be used as arguments to commands. There are four kinds of literals:

Number

Numbers are given as ordinary unsigned decimal numbers. An optional suffix may be used to indicate a multiple of a power of two. The suffixes are: 'K' specifying "kibi-", or 1,024 (2^10) times the value of the number; 'M' specifying "mebi-", or 1,048,576 (2^20) times the value of the number; and 'G' specifying "tebi-", or 1,073,741,824 (2^30) times the value of the number.

The numbers have 32 bits of magnitude.

String

A *string* is any sequence of characters enclosed in double quotes ('"'). A string cannot contain newlines and double quote characters. This limitation will disappear in future releases.

Multiline Strings

A *multiline string* is used to represent large blocks of text with embedded newlines and special characters. It starts with the keyword `text:` followed by a newline and ends with a dot ('.') on a newline by itself. Any characters between these two markers are taken verbatim. For example:

```
text:
** This is an authomatic response from my message **
** filtering program.                               **

I can not attend your message right now. However it
will be saved, and I will read it as soon as I am back.

Regards,
Fred
.
```

Notice that a hashed comment or whitespace may occur between `text:` and the newline. However, when used inside the multiline string a hash sign looses its special meaning (except in one case, see below) and is taken as is, as well as bracketed comment delimiters. In other words, no comments are allowed within a multiline string. E.g.:

```
text: # This is a comment

Sample text
# This line is taken verbatim
/* And this line too */
.
```

The only exception to this rule is that preprocessor `include` statement is expanded as usual when found within a multiline string (see Section 5.2.3.1 [#include], page 58), e.g.:

```
text:
#include <myresponse.txt>
.
```

This results in the contents of file '`myresponse.txt`' being read and interpreted as the contents of the multiline string.

String Lists

A *string list* is a comma-delimited list of quoted strings, enclosed in a pair of square brackets, e.g.:

```
["me@example.com", "me00@landru.example.edu"]
```

For convenience, in any context where a list of strings is appropriate, a single string is allowed without being a member of a list: it is equivalent to a list with a single member. For example, the following two statements are equivalent:

```
exists "To";
exists ["To"];
```

5.2.2 Syntax

Being designed for the sole purpose of filtering mail, Sieve has a very simple syntax.

5.2.2.1 Commands

The basic syntax element is a *command*. It is defined as follows:

command-name [*tags*] *args*

where *command-name* is an identifier representing the name of the command, *tags* is an optional list of *optional* or *tagged arguments* and *args* is a list of *required* or *positional arguments*.

Positional arguments are literals delimited with whitespace. They provide the command with the information necessary to its proper functioning. Each command has a fixed number of positional arguments. It is an error to supply more arguments to the command or to give it fewer arguments than it accepts.

Optional arguments allow to modify the behaviour of the command, like command line options in UNIX do. They are a list of *tags* (see Section 5.2.1 [Lexical Structure], page 54) separated by whitespace. An optional argument may have at most one parameter.

Each command understands a set of optional arguments. Supplying it tags that it does not understand results in an error.

For example, consider the following command

```
header :mime :comparator "i;octet" ["to", "from"] "bug-mailutils@gnu.org"
```

Here, given that `header` takes two positional arguments: `header` is command name, the list `["to", "from"]` is first positional argument and the string `"bug-mailutils@gnu.org"` is second positional argument. There are two optional arguments: `:mime` and `:comparator`. The latter has a string `"i;octet"` as its parameter.

5.2.2.2 Actions Described

An *action* is a Sieve command that performs some operation over the message. Actions do the main job in any Sieve program. Syntactically, an action is a command terminated with semicolon, e.g.:

```
keep;

fileinto "mbox";
```

GNU Sieve provides the full set of actions described in RFC 3028. It also allows to extend this set using loadable actions. See Section 5.2.7 [Actions], page 65, for detailed discussion of actions.

5.2.2.3 Control Flow

The only control flow statement Sieve has is "if" statement. In its simplest form it is:

```
if condition { ... }
```

The effect of this statement is that the sequence of actions between the curly braces is executed only if the `condition` evaluates to `true`.

A more elaborate form of this statement allows to execute two different sets of actions depending on whether the condition is true or not:

```
if condition { ... } else { ... }
```

The most advanced form of the "if" statement allows to select an action depending on what condition from the set of conditions is met.

```
if cond1 { ... } elsif cond2 { ... } else { ... }
```

There may be any number of "elsif" branches in an "if" statement. However it may have at most one "else" branch. Notes for C programmers:

1. The braces surrounding each branch of an "if" statement are required.
2. The "else if" construct is disallowed. Use "elsif" keyword instead.

Here's an example of "if" statement:

```
if header :contains "from" "coyote"
  {
    discard;
  }
elsif header :contains ["subject"] ["$$$"]
  {
    discard;
  }
else
  {
```

```
    fileinto "INBOX";
  }
```

The following section describes in detail conditions used in "if" statements.

5.2.2.4 Tests and Conditions

Tests are Sieve commands that return boolean value. E.g. the test

```
header :contains "from" "coyote"
```

returns true only if the header "From" of the current message contains substring "coyote".

The tests shipped with the GNU Sieve are described in Section 5.2.6 [Tests], page 60.

Condition is a Sieve expression that evaluates to `true` or `false`. In its simplest form, condition is just a Sieve test.

To reverse the sense of a condition use keyword `not`, e.g.:

```
not header :contains "from" "coyote"
```

The results of several conditions may be joined together by logical `and` and `or` operations. The special form `allof` takes several tests as its arguments and computes the logical `and` of their results. Similarly, the form `anyof` performs logical `or` over the results of its arguments. E.g.:

```
if anyof (not exists ["From", "Date"],
          header :contains "from" "fool@example.edu")
  {
    discard;
  }
```

5.2.3 Preprocessor

The preprocessor statements are a GNU extension to the Sieve language. The syntax for a preprocessor statement is similar to that used in `C` programming language, i.e.: a pound character ('`#`') followed by a preprocessor directive and its arguments. Any amount of whitespace can be inserted between the '`#`' and the directive. Currently implemented directives are `include` and `searchpath`.

5.2.3.1 Sieve #include directive

The `#include` directive reads in the contents of the given file. The contents is "inserted" into the text being parsed starting at the line where the directive appears. The directive takes two forms:

`#include "`*filename*`"`
: The *filename* is taken relative to the current directory.

`#include <`*filename*`>"`
: The *filename* is searched in the list of include directories as specified by the '`-I`' command line options.

If *filename* starts with a directory separator character ('`/`') both forms have the same effect.

5.2.3.2 Sieve #searchpath directive

The `#searchpath` directive adds its argument to the list of directories searched for loadable modules. It has the same effect as '`-L`' command line switch used by GNU sieve utility (see Section 6.2.8 [sieve group], page 77).

5.2.4 Require Statement

```
Syntax:   require string;
          require string-list;
```

The require statement informs the parser that a script makes use of a certain extension. Multiple capabilities can be declared using the second form of the statement. The actual handling of a capability name depends on its suffix.

If the name starts with '`comparator-`', it is understood as a request to use the specified comparator. The comparator name consists of the characters following the suffix.

If the name starts with '`test-`', it means a request to use the given test. The test name consists of the characters following the suffix.

Otherwise, the capability is understood as a name of an action to be used.

The `require` statement, if present, must be used before any other statement that is using the required capability. As an extension, the GNU sieve allows the `require` and any other statements to be interspersed.

By default the following actions and comparators are always required:

stop

keep

discard

i;octet

i;ascii-casemap

Example:

```
require ["fileinto", "reject"];

require "fileinto";

require "comparator-i;ascii-numeric";
```

When processing arguments for `require` statement, GNU libsieve uses the following algorithm:

1. Look up the name in a symbol table. If the name begins with '`comparator-`' it is looked up in the comparator table. If it begins with '`test-`', the test table is used instead. Otherwise the name is looked up in the action table.
2. If the name is found, the search is terminated.
3. Otherwise, transform the name. First, any '`comparator-`' or '`test-`' prefix is stripped. Then, any character other than alphanumeric characters, '`.`' and '`,`' is replaced with dash ('`-`'). The name thus obtained is used as a file name of an external loadable module.

4. Try to load the module. The module is searched in the following search paths (in the order given):
   1. Mailutils module directory. By default it is '`$prefix/lib/mailutils`'.
   2. Sieve library path as given with the '`-L`' options in the command line
   3. Additional search directories specified with the `#searchpath` directive.
   4. The value of the environment variable LTDL_LIBRARY_PATH.
   5. System library search path: The system dependent library search path (e.g. on Linux it is set by the contents of the file '`/etc/ld.so.conf`' and the value of the environment variable LD_LIBRARY_PATH).

   The value of LTDL_LIBRARY_PATH and LD_LIBRARY_PATH must be a colon-separated list of absolute directories, for example, '`"/usr/lib/mypkg:/lib/foo"`'.

   In any of these directories, `libsieve` first attempts to find and load the given filename. If this fails, it tries to append the following suffixes to the file name:
   1. the libtool archive extension '`.la`'
   2. the extension used for native dynamic libraries on the host platform, e.g., '`.so`', '`.sl`', etc.
5. If the module is found, `libsieve` executes its initialization function (see below) and again looks up the name in the symbol table. If found, search terminates successfully.
6. If either the module is not found, or the symbol wasn't found after execution of the module initialization function, search is terminated with an error status. `libsieve` then issues the following diagnostic message:

   ```
   source for the required action NAME is not available
   ```

5.2.5 Comparators

GNU libsieve supports the following built-in comparators:

`i;octet` This comparator simply compares the two arguments octet by octet

`i;ascii-casemap`
: It treats uppercase and lowercase characters in the ASCII subset of UTF-8 as the same. This is the default comparator.

`i;ascii-numeric`
: Treats the two arguments as ASCII representation of decimal numbers and compares their numeric values. This comparator must be explicitly required prior to use.

5.2.6 Tests

This section describes the built-in tests supported by GNU libsieve. In the discussion below the following macro-notations are used:

match-type
: This tag specifies the matching type to be used with the test. It can be one of the following:

`:is` The `:is` match type describes an absolute match; if the contents of the first string are absolutely the same as the contents of the second string, they match. Only the string "frobnitzm" is the string "frobnitzm". The null key ":is" and only ":is" the null value. This is the default match-type.

`:contains`
The `:contains` match type describes a substring match. If the value argument contains the key argument as a substring, the match is true. For instance, the string "frobnitzm" contains "frob" and "nit", but not "fbm". The null key "" is contained in all values.

`:matches` The `:matches` version specifies a wildcard match using the characters '`*`' and '`?`'. '`*`' matches zero or more characters, and '`?`' matches a single character. '`?`' and '`*`' may be escaped as '`\\?`' and '`\\*`' in strings to match against themselves. The first backslash escapes the second backslash; together, they escape the '`*`'.

`:regex` The `:regex` version specifies a match using POSIX Extended Regular Expressions.

comparator

A *comparator* syntax item is defined as follows:

```
:comparator "comparator-name"
```

It instructs sieve to use the given comparator with the test. If *comparator-name* is not one of '`i;octet`', '`i;ascii-casemap`' it must be required prior to using it. For example:

```
require "comparator-i;ascii-numeric";

if header :comparator "i;ascii-numeric" :is "X-Num" "10"
  {
     ...
```

address-part

This syntax item is used when testing structured Internet addresses. It specifies which part of an address must be used in comparisons. Exactly one of the following tags may be used:

`:all` Use the whole address. This is the default.

`:localpart`
Use local part of the address.

`:domain` Use domain part of the address.

Notice, that *match-type* modifiers interact with comparators. Some comparators are not suitable for matching with `:contains` or `:matches`. If this occurs, sieve issues an appropriate error message. For example, the statement:

```
if header :matches :comparator "i;ascii-numeric"
```

would result in the following error message:

```
comparator 'i;ascii-numeric' is incompatible with match type ':matches'
in call to 'header'
```

false [Test]

This test always evaluates to "false".

true [Test]

This test always evaluates to "true".

address [*address-part*][*comparator*][*match-type*] *header-names key-list* [Test]

Tagged arguments:

address-part
: Selects the address part to compare. Default is the whole email address (`:all`).

comparator
: Specifies the comparator to be used instead of the default `i;ascii-casemap`.

match-type
: Specifies the match type to be used instead of the default `:is`.

Required arguments:

header-names
: A list of header names.

key-list
: A list of address values.

The `address` test matches Internet addresses in structured headers that contain addresses. It returns `true` if any header contains any key in the specified part of the address, as modified by *comparator* and *match-type* optional arguments.

This test returns `true` if any combination of the *header-names* and *key-list* arguments match.

The `address` primitive never acts on the phrase part of an email address, nor on comments within that address. Use the `header` test instead. It also never acts on group names, although it does act on the addresses within the group construct.

Example:

```
if address :is :all "from" "tim@example.com"
  {
     discard;
  }
```

size [*:over*|*:under*] *number* [Test]

The `size` test deals with the size of a message. The required argument *number* represents the size of the message in bytes. It may be suffixed with the following quantifiers:

'`k`'
'`K`'
: The number is expressed in kilobytes.

'`m`'
'`M`'
: The number is expressed in megabytes.

'g'

'G' The number is expressed in gigabytes.

If the tagged argument is ':over', and the size of the message is greater than *number*, the test is true; otherwise, it is false.

If the argument is ':under', and the size of the message is less than the *number*, the test is true; otherwise, it is false.

Otherwise, the test is true only if the size of the message equals exactly *number*. This is a GNU extension.

The size of a message is defined to be the number of octets from the initial header until the last character in the message body.

envelope [*address-part*][*comparator*][*match-type*] *envelope-part key-list* [Test]

Tagged arguments:

address-part
Selects the address part to compare. Default is the whole email address (:all).

comparator
Specifies the comparator to be used instead of the default i;ascii-casemap.

match-type
Specifies the match type to be used instead of the default :is.

Required arguments:

envelope-parts
A list of envelope parts to operate upon.

key-list A list of address values.

The envelope test is true if the specified part of the SMTP envelope matches the specified key.

If the envelope-part strings is (case insensitive) 'from', then matching occurs against the FROM address used in the SMTP MAIL command.

Notice, that due to the limitations imposed by SMTP envelope structure the use of any other values in *envelope-parts* header is meaningless.

exists *header-names* [Test]

Required arguments:

header-names
List of message header names.

The exists test is true if the headers listed in *header-names* argument exist within the message. All of the headers must exist or the test is false.

The following example throws out mail that doesn't have a From header and a Date header:

```
if not exists ["From","Date"]
  {
     discard;
  }
```

header [*comparator*] [*match-type*] [*:mime*] *header-names key-list* [Test]

Tagged arguments:

comparator
: Specifies the comparator to be used instead of the default `i;ascii-casemap`.

match-type
: Specifies the match type to be used instead of the default `:is`.

:mime
: This tag instructs `header` to search through the mime headers in multipart messages as well.

Required arguments:

header-names
: A list of header names.

key-list
: A list of header values.

The `header` test evaluates to true if any header name matches any key. The type of match is specified by the optional match argument, which defaults to ":is" if not explicitly given.

The test returns `true` if any combination of the *header-names* and *key-list* arguments match.

If a header listed in *header-names* exists, it contains the null key ('""'). However, if the named header is not present, it does not contain the null key. So if a message contained the header

```
X-Caffeine: C8H10N4O2
```

these tests on that header evaluate as follows:

```
header :is ["X-Caffeine"] [""] ⇒ false
header :contains ["X-Caffeine"] [""] ⇒ true
```

numaddr [*:over*|*:under*] *header-names number* [Test]

This test is provided as an example of loadable extension tests. You must use '`require "test-numaddr"`' statement before actually using it.

The `numaddr` test counts Internet addresses in structured headers that contain addresses. It returns true if the total number of addresses satisfies the requested relation.

If the tagged argument is '`:over`' and the number of addresses is greater than *number*, the test is true; otherwise, it is false.

If the tagged argument is '`:under`' and the number of addresses is less than *number*, the test is true; otherwise, it is false.

If the tagged argument is not given, '`:over`' is assumed.

5.2.7 Actions

The GNU libsieve supports the following default actions:

stop

keep

discard

fileinto

reject

redirect

Among them the first three actions do not need to be explicitly required by a `require` statement, while the others do.

These actions are described in detail below.

stop [Action]

The `stop` action ends all processing. If no actions have been executed, then the `keep` action is taken.

keep [Action]

The effect of this action is to preserve the current message in the mailbox. This action is executed if no other action has been executed.

discard [Action]

`Discard` silently throws away the current message. No notification is returned to the sender, the message is deleted from the mailbox.

Example:

```
if header :contains ["from"] ["idiot@example.edu"]
  {
    discard;
  }
```

fileinto *folder* [Action]

Required arguments:

folder A string representing the folder name

The `fileinto` action delivers the message into the specified folder.

reject *reason* [Action]

The optional `reject` action refuses delivery of a message by sending back a message delivery notification to the sender. It resends the message to the sender, wrapping it

in a "reject" form, noting that it was rejected by the recipient. The required argument *reason* is a string specifying the reason for rejecting the message.

Example:

If the message contained

```
Date: Tue, 1 Apr 1997 09:06:31 -0800 (PST)
From: coyote@desert.example.org
To: roadrunner@acme.example.com
Subject: I have a present for you

I've got some great birdseed over here at my place.
Want to buy it?
```

and the user's script contained:

```
if header :contains "from" "coyote@desert.example.org"
  {
    reject "I am not taking mail from you, and I don't want
            your birdseed, either!";
  }
```

then the original sender <coyote@desert.example.org> would receive the following notification:

```
To: <coyote@desert.example.org>
X-Authentication-Warning: roadrunner set sender using -f flag
Content-Type: multipart/mixed; boundary=----- =_aaaaaaaaaa0
MIME-Version: 1.0
----- =_aaaaaaaaaa0
The original message was received at
Tue, 1 Apr 1997 09:07:15 -0800 from
coyote@desert.example.org.
Message was refused by recipient's mail filtering program.
Reason given was as follows:

I am not taking mail from you, and I don't want your birdseed, either!

----- =_aaaaaaaaaa0
Content-Type: message/delivery-status

Reporting-UA: sieve; GNU Mailutils 0.1.3
Arrival-Date: Tue, 1 Apr 1997 09:07:15 -0800
Final-Recipient: RFC822; roadrunner@acme.example.com
Action: deleted
Disposition: automatic-action/MDN-sent-automatically;deleted
Last-Attempt-Date: Tue, 1 Apr 1997 09:07:15 -0800

----- =_aaaaaaaaaa0
Content-Type: message/rfc822

From: coyote@desert.example.org
```

```
To: roadrunner@acme.example.com
Subject: I have a present for you

I've got some great birdseed over here at my place.
Want to buy it?
----- =_aaaaaaaaaa0
```

If the *reason* argument is rather long, the common approach is to use the combination of the `text:` and `#include` keywords, e.g.:

```
if header :mime :matches "Content-Type"
          [ "*application/msword;*", "*audio/x-midi*" ]
  {
    reject text:
#include "nomsword.txt"
    .
    ;
  }
```

redirect *address* [Action]

The `redirect` action is used to send the message to another user at a supplied *address*, as a mail forwarding feature does. This action makes no changes to the message body or existing headers, but it may add new headers. It also modifies the envelope recipient.

The `redirect` command performs an MTA-style "forward" — that is, what you get from a '`.forward`' file using `sendmail` under UNIX. The address on the SMTP envelope is replaced with the one on the `redirect` command and the message is sent back out. *Notice*, that it differs from the MUA-style forward, which creates a new message with a different sender and message ID, wrapping the old message in a new one.

5.3 Extensions

This section summarizes the GNU extensions to the sieve language

1. Handling of the `require` statement.

   According to the RFC an error must occur if a `require` appears after a command other than `require`. The GNU sieve library allows interspersing the `require` and other statements. The only requirement is that `require` must occur before a statement that is using the required capability (see Section 5.2.4 [Require Statement], page 59).

   Prefixing the required capability with "test" requires the use of an extension test.

2. `header` test

   The `header` takes an optional argument `:mime`, meaning to scan the headers from each part of a multipart message.

3. `size` test

   The `size` test allows to omit the optional argument (:over|:under). In this case exact equality is assumed.

4. `envelope` test

   The only value that can be meaningfully used as the first required argument of an `envelope` test is '`from`'. This limitation may disappear from the subsequent releases.

5. Match type optional argument.

   Along with the usual `:is`, `:matches` and `contains` matching type, GNU sieve library understands `:regex` type. This matching type toggles POSIX Extended Regular Expression matching.

5.4 Writing Loadable Commands

This section contains an example of how to write external loadable commands for GNU libsieve.

```
/*  This is an example on how to write extension tests for GNU sieve.
    It provides test "numaddr".

    Syntax:   numaddr [":over" / ":under"] <header-names: string-list>
              <limit: number>

    The "numaddr" test counts Internet addresses in structured headers
    that contain addresses.  It returns true if the total number of
    addresses satisfies the requested relation:

    If the argument is ":over" and the number of addresses is greater than
    the number provided, the test is true; otherwise, it is false.

    If the argument is ":under" and the number of addresses is less than
    the number provided, the test is true; otherwise, it is false.

    If the argument is empty, ":over" is assumed. */

#ifdef HAVE_CONFIG_H
# include <config.h>
#endif

#include <stdlib.h>
#include <mailutils/libsieve.h>

struct val_ctr {  /* Data passed to the counter function */
  header_t hdr;   /* Headers of the current message */
  size_t limit;   /* Limit for the number of addresses */
  size_t count;   /* Number of addresses counted so far */
};

/* Count addresses in a single header value.

   Input:
     ITEM is the name of the header to scan.
```

```
     DATA is a pointer to the val_ctr structure
   Return value:
     non-zero if the limit on the number of addresses has been reached. */

static int
_count_items (void *item, void *data)
{
  char *name = item;
  struct val_ctr *vp = data;
  char *val;
  address_t addr;
  size_t count = 0;

  if (header_aget_value (vp->hdr, name, &val))
    return 0;

  if (address_create (&addr, val) == 0)
    {
      address_get_count (addr, &count);
      address_destroy (&addr);
      vp->count += count;
    }
  free (val);
  return vp->count >= vp->limit;
}

/* Handler for the numaddr test */
static int
numaddr_test (sieve_machine_t mach, list_t args, list_t tags)
{
  sieve_value_t *h, *v;
  struct val_ctr vc;
  int rc;

  if (sieve_get_debug_level (mach) & MU_SIEVE_DEBUG_TRACE)
    sieve_debug (mach, "NUMADDR\n");

  /* Retrieve required arguments: */
  /* First argument: list of header names */
  h = sieve_value_get (args, 0);
  if (!h)
    {
      sieve_error (mach, "numaddr: can't get argument 1");
      sieve_abort (mach);
    }
  /* Second argument: Limit on the number of addresses */
  v = sieve_value_get (args, 1);
  if (!v)
    {
```

```
      sieve_error (mach, "numaddr: can't get argument 2");
      sieve_abort (mach);
    }

  /* Fill in the val_ctr structure */
  message_get_header (sieve_get_message (mach), &vc.hdr);
  vc.count = 0;
  vc.limit = v->v.number;

  /* Count the addresses */
  rc = sieve_vlist_do (h, _count_items, &vc);

  /* Here rc >= 1 iff the counted number of addresses is greater or equal
     to vc.limit. If `:under' tag was given we reverse the return value */
  if (sieve_tag_lookup (tags, "under", NULL))
    rc = !rc;
  return rc;
}

/* Syntactic definitions for the numaddr test */

/* Required arguments: */
static sieve_data_type numaddr_req_args[] = {
  SVT_STRING_LIST,
  SVT_NUMBER,
  SVT_VOID
};

/* Tagged arguments: */
static sieve_tag_def_t numaddr_tags[] = {
  { "over", SVT_VOID },
  { "under", SVT_VOID },
  { NULL }
};

static sieve_tag_group_t numaddr_tag_groups[] = {
  { numaddr_tags, NULL },
  { NULL }
};

/* Initialization function. It is the only function exported from this
   module. */
int
SIEVE_EXPORT(numaddr,init) (sieve_machine_t mach)
{
  return sieve_register_test (mach, "numaddr", numaddr_test,
                              numaddr_req_args, numaddr_tag_groups, 1);
}
```

6 Programs

GNU Mailutils provides a set of programs for handling the email.

6.1 Authorization and authentication principles.

Some mail utilities provide access to their services only after verifying that the user is actually the person he is claiming to be. Such programs are, for example, `pop3d` and `imap4d`. The process of the verification is broken down into two stages: *authorization* and *authentication*. In *authorization* stage the program retrieves the information about a particular user. In *authentication* stage, this information is compared against the user-supplied credentials. Only if both stages succeed is the user allowed to use the service.

A set of *modules* is involved in performing each stage. For example, the authorization stage can retrieve the user description from various sources: system database, sql database, virtual domain table, etc. Each module is responsible for retrieving the description from a particular source of information. The modules are arranged in a *module list*. The modules from the list are invoked in turn, until either a one of them succeeds or the list is exhausted. In latter case the authorization fails. Otherwise the data returned by the succeeded module are used in authentication.

Similarly, authentication may be performed in several ways. The authentication modules are also grouped in a list. Each module is tried in turn until either a module succeeds, in which case the authentication succeeds, or the end of the list is reached.

We represent the module lists as column-separated lists of module names. For example, the authorization list

```
system:sql:virtdomains
```

means that first the system user database (‘`/etc/password`’) is searched for a description of a user in question. If the search fails, the SQL database is searched. Finally, if it also fails, the search is performed in the virtual domain database.

Note, that some authentication and/or authorization modules may be disabled when configuring the package before compilation. The names of the disabled modules are nevertheless available for use in runtime configuration options, but they represent a “fail-only” functionality, e.g. if the package was compiled without SQL support then the module ‘`sql`’ in the above example will always fail, thus passing the execution on to the next module.

The modules available for use in authorization list are:

system
: User credentials are retrieved from the system user database (‘`/etc/password`’).

sql
: User credentials are retrieved from the SQL database. The set of ‘`--sql-`’ options (see Section 6.2.6 [auth], page 76) is used to configure access to the database.

virtdomain
: User credentials are retrieved from a “virtual domain” user database.

The modules available for use in authentication list are:

generic
: The generic authentication type. User password is hashed and compared against the hash value returned in authorization stage.

system
: The hashed value of the user password is retrieved from ‘`/etc/shadow`’ file on systems that support it.

sql
: The hashed value of the user password is retrieved from the SQL database using query supplied by ‘`--sql-getpass`’ option (see Section 6.2.6 [auth], page 76).

pam The user is authenticated via pluggable authentication module (PAM). The PAM service name to be used is configured via '`--pam-service`' option (see Section 6.2.6 [auth], page 76)

Unless overridden by '`--authentication`' command line option, the list of authentication modules is:

```
generic:system:pam:sql
```

Unless overridden by '`--authorization`' command line option, the list of authorization modules is:

```
system:sql:virtdomains
```

6.2 Mailutils configuration file

There are some command line options that are used so often that it is inconvenient to specify them in the command line each time you run a Mailutils utility. The *configuration files* provide a way to add default command line arguments without having to type them in the command line. Upon startup, each Mailutils utility scans and processes the contents of the three startup files, none of which are required to exist:

1. the site-wide configuration file

   '`mailutils.rc`', found in your your system configuration directory (usually '`/etc`' or '`/usr/local/etc`').

2. the user-specific configuration file

   Usually '`~/.mailutils`', unless '`~/.mailutils`' is a directory, in which case '`~/.mailutils/mailutils`' is used.

3. the programs-specific configuration file

   Usually '`~/.mu.`*program*`rc`', unless '`~/.mailutils`' is a directory, in which case '`~/.mailutils/`*program*`rc`' is used (where *program* means the program name).

These files have simple line-oriented syntax. Comments begin with the pound sign ('`#`') and extend through the end of the line[1]. Very long lines may be split across several lines by escaping final newline with a backslash ('\') character.

In the non-program-specific configuration files, any configuration line must start with a *tag*. In the program-specific configuration file the tag must not be present, all options are for that specific program.

A tag is either a name of a particular Mailutils utility or *option group*, prefixed with colon (':'). The command line options common for several Mailutils programs are divided into *option groups* or *capabilities*, e.g. the options '`--mail-spool`' and '`--lock-flags`' form group '`mailbox`'. These groups are discussed in detail below.

When processing the non-program-specific configuration files, a Mailutils utility selects those lines whose tag is either the name of that utility or the name of the option group supported by it. In the program-specific configuration file, all lines are selected. For each line found, its tag (if present) is stripped away, and the rest of the line is split up into words. These words are regarded as command line options and are inserted to the program arguments *before* any options from the command line. Thus the options from '`.mailutils`' take precedence over those from '`mailutils.rc`', and the options from the command line take precedence over those from all three configuration files.

The word splitting occurs at whitespace characters and is similar to that performed by the shell. If an option must contain embedded whitespace, it should be enclosed in a pair of quotes (either double or single quotes).

6.2.1 default — Options understood by most GNU utilities.

Each program also understands the following informational options:

[1] If '`#`' is not the first character on the line, it should be separated from the previous word by any amount of whitespace.

'`-u`'
'`--usage`' Display a short usage message and exit.

'`-h`'
'`--help`' Display help message and exit.

'`-L`'
'`--license`'
Display GNU General Public License and exit.

'`-v`'
'`--version`'
Display program version and exit.

6.2.2 mailbox — Specifies the mail spool location, and locking strategy.

Option group '`mailbox`' consists of options used to specify the location of the mail spool, and the locking strategy.

'`-m` *path*'
'`--mail-spool=`*path*'
Set path to the mailspool directory

'`--lock-flags=`*flags*'
Set the default mailbox lock flags (E=external, R=retry, T=time, P=pid).

6.2.3 mailer — Sets the default mailer URL.

This option group overrides the default mailer URL (`sendmail:`).

'`-m` *url*'
'`--mailer` *url*'

6.2.4 address — Specifies the default email address and domain.

Option group '`address`' consists of options used to specify how to qualify email addresses.

An unqualified address (one without an `@`) is qualified by appending `@`*defaultdomain*. *defaultdomain* is the return of gethostname(), or the result of gethostbyname() on the return of gethostname() (if the DNS lookup is successful).

If the email address of the current user is needed, either the address set by –email-addr is returned, or the current uid is looked up in the user database, and qualified with the *defaultdomain*.

'`-E` *email*'
'`--email-addr=`*email*'
Set the current user's email address, this it makes more sense to use this in one of the per-user configuration files.

'`-D` *domain*'
'`--email-domain=`*domain*'
Set the default email domain, this might make sense to use in either the global or one of the per-user configuration files.

6.2.5 daemon — Options common for daemon programs.

'`-d[`*number*`]`'
'`--daemon[=`*number*`]`'
: Run in standalone mode. An optional *number* specifies the maximum number of child processes the daemon is allowed to fork. When it is omitted, it defaults to 20 processes. *Please note*, that there should be no whitespace between the '`-d`' and its parameter.

'`-i`'
'`--inetd`' Run in inetd mode.

'`-p` *number*'
'`--port` *number*'
: Listen on given port *number*. This option is meaningful only in standalone mode. It defaults to port 143.

'`-t` *number*'
'`--timeout` *number*'
: Set idle timeout to given *number* of seconds. The daemon breaks the connection if it receives no commands from the client within that number of seconds.

6.2.6 auth — Authentication-specific options.

These options control the authorization and authentication module lists. For a description of authentication concepts, refer to See Section 6.1 [authentication], page 72.

'`--authorization` *modlist*'
: This option allows to set up a list of modules to be used for authorization. *modlist* is a colon-separated list of modules. Valid modules are:

  system — User credentials are retrieved from the system user database ('`/etc/password`').

  sql — User credentials are retrieved from the SQL database. The set of '`--sql-`' options (see below) is used to configure access to the database.

  virtdomain — User credentials are retrieved from a "virtual domain" user database.

'`--authentication` *modlist*'
: This option allows to set up a list of modules to be used for authentication. *modlist* is a colon-separated list of modules. Valid modules are:

  generic — The generic authentication type. User password is hashed and compared against the hash value returned in authorization stage.

  system — The hashed value of the user password is retrieved from '`/etc/shadow`' file on systems that support it.

  sql — The hashed value of the user password is retrieved from the SQL database using query supplied by '`--sql-getpass`' option (see below).

pam
: The user is authenticated via pluggable authentication module (PAM). The PAM service name to be used is configured via '`--pam-service`' option (see below)

'`--pam-service` *name*'
: When compiled with PAM support, this option specifies the name of PAM service to be used when authenticating.

The following options exist in this group if the package was configured with '`--enable-sql`' option. They take effect only if the '`sql`' module is used in authentication and/or authorization. Currently only MySQL is supported.

'`--sql-getpwnam` *query*'
: SQL query to retrieve a passwd entry based on username

'`--sql-getpwuid` *query*'
'`--sql-getpass` *query*'
: SQL query to retrieve a password from the database

'`--sql-host` *name*'
: Name or IP of MySQL server to connect to.

'`--sql-user` *name*'
: SQL user name

'`--sql-passwd` *string*'
: SQL connection password

'`--sql-db` *string*'
: Name of the database to connect to.

'`--sql-port` *number*'
: Port to use

6.2.7 logging — Logging control options.

'`--log-facility` *facility*'
: Output logs to the specified `syslog` facility. The following facility names are recognized: '`user`', '`daemon`', '`mail`', '`auth`' and '`local0`' through '`local7`'. These names are case-insensitive.

6.2.8 Sieve specific options

The following options comprise this group:

'`-I` *dir*'
'`--includedir=`*dir*'
: Append directory *dir* to the list of directories searched for include files.

'`-L` *dir*'
'`--libdir=`*dir*'
: Append directory *dir* to the list of directories searched for library files.

6.2.9 A sample configuration file.

The following configuration file specifies that all Mailutils programs should use '`/var/spool/mail`' as a local mailspool directory. Programs performing authentication will use PAM service '`mailutils`'. All programs, except `imap4d` will issue log messages via '`mail`' facility, `imap4d` will use facility '`local1`'.

```
:mailbox --mail-spool /var/spool/mail
:auth --authentication pam --pam-service mailutils
:logging --log-facility mail
imap4d --daemon=20 --timeout=1800 --log-facility local1
```

6.3 IMAP4 daemon

GNU imap4d is a daemon implementing IMAP4 rev1 protocol for accessing and handling electronic mail messages on a server. It can be run either as a standalone program or from '`inetd.conf`' file.

6.3.1 Imap4d Namespace

GNU imap4d supports a notion of *namespaces* defined in RFC 2342. A namespace is a set of directories upon which the user has certain permissions. It should be understood that these permissions apply only if the underlying filesystem allows them.

The three namespaces supported by `imap4d` are:

Personal Namespace
: A namespace that is within the personal scope of the authenticated user on a particular connection. The user has all permissions on this namespace.

Other Users' Namespace
: A namespace that consists of mailboxes from the "Personal Namespaces" of other users. The user can read and list mailboxes from this namespace. However, he is not allowed to use '`%`' and '`*`' wildcards with `LIST` command, that is he can access a mailbox only if he knows exactly its location.

Shared Namespace
: A namespace that consists of mailboxes that are intended to be shared amongst users and do not exist within a user's Personal Namespace. The user has all permissions on this namespace.

By default, `imap4d` starts with the following namespaces:

Personal Namespace
: The home directory of the user, if exists.

Other Users' Namespace
: Empty

Shared Namespace
: Empty

Note, that this means that by default, a user won't be able to see or otherwise access mailboxes residing in the directories other than his own home.

To change these defaults, use '`--shared-namespace`' and '`--other-namespace`' options.

6.3.2 Starting imap4d

`imap4d` may run either in *standalone* or in *inetd* operation modes. When run in "standalone" mode, the server disconnects from the terminal and runs as a daemon, forking a child for each new connection.

The "inetd" mode allows to start the server from '`/etc/inetd.conf`' file. This is the default operation mode.

```
imap4  stream tcp nowait  root  /usr/local/sbin/imap4d imap4d
```

The program uses following option groups: See Section 6.2.2 [mailbox], page 75, See Section 6.2.5 [daemon], page 76, See Section 6.2.7 [logging], page 77, See Section 6.2.6 [auth], page 76.

Command line options

'`-d`[*number*]'
'`--daemon`[`=`*number*]'
: Run in standalone mode. An optional *number* specifies the maximum number of child processes the daemon is allowed to fork. When it is omitted, it defaults to 20 processes. *Please note*, that there should be no whitespace between the '`-d`' and its parameter.

'`-h`'
'`--help`'
: Display short help message and exit.

'`-i`'
'`--inetd`'
: Run in inetd mode.

'`-m` *path*'
'`--mail-spool=`*path*'
: Set path to the mailspool directory

'`-O` *pathlist*'
'`--other-namespace=`*pathlist*'
: Set the list of directories forming the "Other User's" namespace. *pathlist* is a list of directory names separated by colons.

'`-p` *number*'
'`--port` *number*'
: Listen on given port *number*. This option is meaningful only in standalone mode. It defaults to port 143.

'`-S` *pathlist*'
'`--shared-namespace=`*pathlist*'
: Set the list of directories, forming the "Shared" namespace. *pathlist* is a list of directory names separated by colons.

'`-t` *number*'
'`--timeout` *number*'
: Set idle timeout to given *number* of seconds. Default is 1800 seconds (30 minutes). The daemon breaks the connection if it receives no commands from the client within that number of seconds.

'`-v`'
'`--version`'
: Display program version and exit.

6.4 POP3 daemon

The `pop3d` daemon implements the Post Office Protocol server.

pop3d has two operation modes:

Inetd
: The server is started from '`/etc/inetd.conf`' file:

```
pop3  stream tcp nowait  root  /usr/local/sbin/pop3d pop3d
```

This is the default operation mode.

Standalone
: The server runs as daemon, forking a child for each new connection. This mode is triggered by '`-d`' command line switch.

The program uses following option groups: See Section 6.2.2 [mailbox], page 75, See Section 6.2.5 [daemon], page 76, See Section 6.2.7 [logging], page 77, See Section 6.2.6 [auth], page 76.

Command line options

'`-d`[*number*]'
'`--daemon`[=*number*]'
: Run in standalone mode. An optional *number* specifies the maximum number of child processes the daemon is allowed to fork. When it is omitted, it defaults to 10 processes. *Please note*, that there should be no whitespace between the '`-d`' and its parameter.

'`-h`'
'`--help`' Display short help message and exit.

'`-i`'
'`--inetd`' Run in inetd mode.

'`-m` *path*'
'`--mail-spool=`*path*'
: Set path to the mailspool directory

'`-p` *number*'
'`--port` *number*'
: Listen on given port *number*. This option is meaningful only in standalone mode. It defaults to port 110.

'`-t` *number*'
'`--timeout` *number*'
: Set idle timeout to given *number* of seconds. Default is 600 seconds (10 minutes). The daemon breaks the connection if it receives no commands from the client within that number of seconds.

'`-v`'
'`--version`'
: Display program version and exit.

6.5 frm — List headers from a mailbox.

The `frm` command outputs a header information of the selected messages in a mailbox. By default, `frm` reads the user's system mailbox and outputs the contents of `From` and `Subject` headers for each message. If a folder is specified in the command line, the program reads that folder rather than the default mailbox.

The program uses following option groups: See Section 6.2.2 [mailbox], page 75.

The following command line options alter the behavior of the program:

'`-f STRING`'
'`--field STRING`'
: Display the header named by STRING instead of `From Subject` pair.

'`-l`'
'`--to`'
: Include the contents of `To` header to the output. The output field order is then: `To From Subject`.

'`-n`'
'`--number`'
: Prefix each line with corresponding message number.

'`-Q`'
'`--Quiet`'
: Be very quiet. Nothing is output except error messages. This is useful in shell scripts where only the return status of the program is important.

'`-q`'
'`--query`'
: Print a message only if there are unread messages in the mailbox.

'`-S`'
'`--summary`'
: Print a summary line.

'`-s ATTR`'
'`--status ATTR`'
: Only display headers from messages with the given status. ATTR may be one of the following: '`new`', '`read`', '`unread`'. It is sufficient to specify only first letter of an ATTR. Multiple '`-s`' options are allowed.

'`-t`'
'`--align`'
: Tidy mode. Currently is not implemented. Included for compatibility with `frm` program from Elm package.

6.6 mail — send and receive mail.

`Mail` is an enhanced version of standard `/bin/mail` program. As well as its predecessor, it can be used either in sending mode or in reading mode. `Mail` enters sending mode when one or more email addresses were specified in this command line. In this mode the program waits until user finishes composing the message, then attempts to send it to the specified addresses and exits. See Section 6.6.3 [Composing Mail], page 86, for a detailed description of this behavior.

If the command line contained no email addresses, `mail` switches to reading mode. In this mode it allows to read and manipulate the contents of a mailbox. The URL of the mailbox to operate upon is taken from the argument of '`--file`' command line option. If it is not specified, the user's system mailbox is assumed. For more detail, see Section 6.6.4 [Reading Mail], page 88.

6.6.1 Command line options

General usage of `mail` program is:

```
mail [OPTION...] [address...]
```

If [address...] part is present, `mail` switches to mail sending mode, otherwise it operates in mail reading mode.

The program uses following option groups: See Section 6.2.2 [mailbox], page 75.

`Mail` understands following command line options:

'`-e`'
'`--exist`' Return true if the mailbox contains some messages. Return false otherwise. This is useful for writing shell scripts.

'`-f[FILE]`'
'`--file[=FILE]`'
Operate on mailbox FILE. If this option is not specified, the default is user's system mailbox. If it is specified without argument, the default is $HOME/mbox. *Please note*, that there should be no whitespace between the short variant of the option ('`-f`'), and its parameter. Similarly, when using long option ('`--file`'), its argument must be preceded by equal sign.

'`-F`'
'`--byname`'
Save messages according to sender. Currently this option is not implemented.

'`-H`'
'`--headers`'
Print header summary to stdout and exit.

'`-i`'
'`--ignore`'
Ignore interrupts.

'`-m` *path*'
'`--mail-spool=`*path*'
Set path to the mailspool directory

'`-n`'
'`--norc`' Do not read the system-wide mailrc file. See Section 6.6.6 [Mail Configuration Files], page 101.

'`-N`'
'`--nosum`' Do not display initial header summary.

'`-p`'
'`--print`'
'`-r`'
'`--read`' Print all mail to standard output. It is equivalent to issuing following commands after starting '`mail -N`':

```
print *
quit
```

'`-q`'
'`--quit`' Cause interrupts to terminate program.

'`-s SUBJ`'
'`--subject=SUBJ`'
Send a message with a Subject of SUBJ. Valid only in sending mode.

'`-t`'
'`--to`' Switch to sending mode.

'`-u USER`'
'`--user=USER`'
Operate on USER's mailbox. This is equivalent to:

```
mail -f/spool_path/USER
```

with *spool_path* being the full path to your mailspool directory ('`/var/spool/mail`' or '`/var/mail`' on most systems).

'`-?`'
'`--help`' Display a help message.
'`--usage`' Display a short usage summary.

'`-V`'
'`--version`'
Print program version and exit.

6.6.2 How to specify message sets

Many mail commands such as print and delete can be given a *message list* to operate upon. Wherever the message list is omitted, the command operates on the current message.

The *message list* in its simplest form is one of:

. Selects current message. It is equivalent to empty message list.

* Selects all messages in the mailbox.

^ Selects first non-deleted message.

$ Selects last non-deleted message.

In its complex form, the *message list* is a comma or whitespace-separated list of *message specifiers*. A *message specifier* is one of

Message Number
: This specifier addresses the message with the given ordinal number in the mailbox.

Message range
: *Message range* is specified as two message numbers separated by a dash. It selects all messages with the number lying within that range.

Attribute specifier
: An *Attribute specifier* is a colon followed by a single letter. The *Attribute specifier* addresses all messages in the mailbox that have the given attribute. These are the valid attribute specifiers:

  ‘`:d`’ Selects all deleted messages.

  ‘`:n`’ Selects all recent messages, i.e. the messages that have not been neither read not seen so far.

  ‘`:o`’ Selects all messages that have been seen.

  ‘`:r`’ Selects all messages that have been read.

  ‘`:u`’ Selects all messages that have *not* been read.

  ‘`:t`’ Selects all tagged messages (see Section 6.6.4.8 [Marking Messages], page 92).

  ‘`:T`’ Selects all untagged messages.

Header match
: The *header match* is a string in the form:

  `[`*header*`:]/`*string*`/`

  It selects all messages that contain header field *header* matching given *regexp*. If the variable `regexp` is set, the *string* is assumed to be a POSIX regexp. Otherwise, a header is considered to match *string* if the latter constitutes a substring of the former (comparison is case-insensitive).

  If *header*: part is omitted, it is assumed to be ‘`Subject:`’.

Message body match
: The *message body match* is a string in the form:

  `:/`*string*`/`

  It selects all messages whose body matches the string. The matching rules are the same as described under “Header match”.

A *message specifier* can be followed by *message part specifier*, enclosed in a pair of brackets. A *message part specifier* controls which part of a message should be operated upon. It is meaningful only for multipart messages. A *message part specifier* is a comma or whitespace - separated list of part numbers or ranges. Each part number can in turn be *message part specifier*, thus allowing for operating upon multiply-encoded messages.

The following are the examples of valid message lists:

6.6.3 Composing mail

You can compose the message by simply typing the contents of it, line by line. But usually this is not enough, you would need to edit your text, to quote some messages, etc. `Mail` provides these capabilities through *compose escapes*. The *compose escapes* are single-character commands, preceded by special *escape character*, which defaults to '`~`'. The combination `escape character + command` is recognized as a compose escape only if it occurs at the beginning of a line. If the escape character must appear at the beginning of a line, enter it twice. The actual escape character may be changed by setting the value of `escape` mail variable (see Section 6.6.5 [Mail Variables], page 97).

6.6.3.1 Quitting Compose Mode

There are several commands allowing you to quit the compose mode.

Typing the end-of-file character ('`C-D`') on a line alone finishes compose mode and sends the message to its destination. The '`C-D`' character looses its special meaning if `ignoreeof` mail variable is set.

If mail variable `dot` is set, typing dot ('`.`') on a line alone achieves the same effect as '`C-D`' above.

Finally, using '`~.`' escape always quits compose mode and sends out the composed message.

To abort composing of a message without sending it, type interrupt character (by default, '`C-C`') twice. This behavior is disabled when mail variable `ignore` is set. In this case, you can use '`~x`' escape to achieve the same effect.

6.6.3.2 Getting Help on Compose Escapes: ~?

The '`~?`' escape prints on screen a brief summary of the available compose escapes. *Please note*, that '`~h`' escape prompts for changing the header values, and does *not* give help.

6.6.3.3 Editing the Message: ~e and ~v.

If you are not satisfied with the message as it is, you can edit it using a text editor specified either by `EDITOR` or by `VISUAL` environment variables. The '`~e`' uses the former, and '`~v`' uses the latter.

By default both escapes allow you to edit only the body of the message. However, if the `editheaders` variable is set, `mail` will load into the editor the complete text of the message with headers included, thus allowing you to change the headers as well.

6.6.3.4 Modifying the Headers: ~h, ~t, ~c, ~b, ~s

To add new addresses to the list of message recipients, use '`~t`' command, e.g.:

```
~t name1@domain.net name2
```

To add addresses to `Cc` or `Bcc`, use '`~c`' or '`~b`' escapes respectively.

To change the `Subject` header, use '`~s`' escape, e.g.:

```
~s "Re: your message"
```

Finally, to edit all headers, type '`~h`' escape. This will present you with the values of `To`, `Cc`, `Bcc`, and `Subject` headers allowing to edit them with normal text editing commands.

6.6.3.5 Enclosing Another Message: ~m and ~M

If you are sending mail from within mail command mode, you can enclose the contents of any message sent to you by using '`~m`' or '`~M`' commands. Typing '`~m`' alone will enclose the contents of the current message, typing '`~m 12`' will enclose the contents of message #12 and so on.

The '`~m`' uses retained and ignored lists when enclosing headers, the '`~M`' encloses all header fields (see Section 6.6.4.5 [Controlling Header Display], page 90).

In both cases, the contents of `indentprefix` mail variable is prepended to each line enclosed.

6.6.3.6 Adding a File to the Message: ~r and ~d

To append the contents of file *filename* to the message, type

```
~r filename
```

or

```
~< filename
```

The '`~d`' escape is a shorthand for

```
~r dead.letter
```

6.6.3.7 Printing And Saving the Message

The '`~p`' escape types the contents of the message entered so far, including headers, on your terminal. You can save the message to an arbitrary file using '`~w`' escape. It takes the filename as its argument.

6.6.3.8 Signing the Message: ~a and ~A

To save you the effort of typing your signature at the end of each message, you can use '`~a`' or '`~A`' escapes. If your signature occupies one line only, save it to the variable `sign` and use '`~a`' escape to insert it. Otherwise, if it is longer than one line, save it to a file, store the name of this file in the variable `Sign`, and use '`~A`' escape to insert it into the message.

6.6.3.9 Printing Another Message: ~f and ~F

Sometimes it is necessary to view the contents of another message, while composing. These two escapes allow it. Both take the message list as their argument. If they are used without argument, the contents of the current message is printed. The difference between '`~f`' and '`~F`' is that the former uses ignored and retained lists to select headers to be displayed, whereas the latter prints all headers (see Section 6.6.4.5 [Controlling Header Display], page 90).

6.6.3.10 Inserting Value of a Mail Variable: ~i

The '`~i`' escape enters the value of the named mail variable into the body of the message being composed.

6.6.3.11 Executing Other Mail Commands: ~: and ~-

You can execute a mail command from within compose mode using '`~:`' or '`~-`' escapes. For example, typing

```
~: from :t
```

will display the from lines of all tagged messages. Note, that executing mail-sending commands (see Section 6.6.4.14 [Replying], page 96) from within the compose mode is not allowed. An attempt to execute such a command will result in diagnostic message "Command not allowed in an escape sequence" being displayed. Also, when starting compose mode immediately from the shell (e.g. running '`mail address@domain`'), most mail commands are meaningless, since there is no mailbox to operate upon. In this case, the only commands that can reasonably be used are: `alias`, `unalias`, `alternate`, `set`, and `unset`.

6.6.3.12 Executing Shell Commands: ~! and ~|

The '`~!`' escape executes specified command and returns you to `mail` compose mode without altering your message. When used without arguments, it starts your login shell. The '`~|`' escape pipes the message composed so far through the given shell command and replaces the message with the output the command produced. If the command produced no output, `mail` assumes that something went wrong and retains the old contents of your message.

6.6.4 Reading mail

To read messages from a given mailbox, use one of the following ways of invoking `mail`:

`mail` To read messages from your system mailbox.

`mail --file`
: To read messages from your mailbox ($HOME/mbox).

`mail --file=`*path_to_mailbox*
: To read messages from the specified mailbox.

`mail --user=`*user*
: To read messages from the system mailbox belonging to *user*.

Please note, that usual mailbox permissions won't allow you to use the last variant of invocation, unless you are a super-user. Similarly, the last but one variant is also greatly affected by the permissions the target mailbox has.

Unless you have started mail with '`--norc`' command line option, it will read the contents of the system-wide configuration file. Then it reads the contents of user configuration file, if any. For detailed description of these files, see Section 6.6.6 [Mail Configuration Files], page 101. After this initial setup, `mail` displays the first page of header lines and enters interactive mode. In interactive mode, `mail` displays its prompt ('`?`', if not set otherwise) and executes the commands the user enters.

6.6.4.1 Quitting the program

Following commands quit the program:

'`quit`' Terminates the session. If `mail` was operating upon user's system mailbox, then all undeleted and unsaved messages that have been read and are not marked with hold flag are saved to the user's mbox file ('`$HOME/mbox`'). The messages, marked with `delete` are removed. The program exits to the Shell, unless saving the mailbox fails, in which case user can escape with the exit command.

'`exit`'
'`ex`'
'`xit`' Program exits to the Shell without modifying the mailbox it operates upon.

Typing EOF ('`C-D`') alone is equivalent to '`quit`'.

6.6.4.2 Obtaining online help

Following commands can be used during the session to request online help:

'`help` [*command*]'
'`hel` [*command*]'
'`?` [*command*]'
Display detailed command synopsis. If no *command* is given, help for all available commands is displayed.

'`list`'
'`*`' Print a list of available commands.

'`version`'
'`ve`' Display program version.

'`warranty`'
'`wa`' Display program warranty statement.

6.6.4.3 Moving within a mailbox

'`next`'
'`n`' Move to the next message.

'`previous`'
'`prev`' Move to the previous message.

6.6.4.4 Changing mailbox/directory

'`cd` [*dir*]'
'`chdir` [*dir*]'
'`ch` [*dir*]' Change to the specified directory. If *dir* is omitted, $HOME is assumed.

'`file` [*mailbox*]'
'`fi` [*mailbox*]'
'`folder` [*mailbox*]'
'`fold` [*mailbox*]'
Read in the contents of the specified *mailbox*. The current mailbox is updated as if `quit` command has been issued. If *mailbox* is omitted, the command prints

the current mailbox name followed by the summary information regarding it, e.g.:

```
& fold
"/var/spool/mail/gray": 23 messages 22 unread
```

6.6.4.5 Controlling header display

To control which headers in the message should be displayed, `mail` keeps two lists: a *retained* header list and an *ignored* header list. If *retained* header list is not empty, only the header fields listed in it are displayed when printing the message. Otherwise, if *ignored* header list is not empty, only the headers *not listed* in this list are displayed. The uppercase variants of message-displaying commands can be used to print all the headers.

The following commands modify and display the contents of both lists.

'`discard` [*header-field-list*]'
'`di` [*header-field-list*]'
'`ignore` [*header-field-list*]'
'`ig` [*header-field-list*]'
: Add *header-field-list* to the ignored list. When used without arguments, this command prints the contents of ignored list.

'`retain` [*header-field-list*]'
'`ret` [*header-field-list*]'
: Add *header-field-list* to the retained list. When used without arguments, this command prints the contents of retained list.

6.6.4.6 Displaying information

'`=`' Displays the current message number.

'`headers` [*msglist*]'
'`h` [*msglist*]'
: Lists the current pageful of headers.

'`from` [*msglist*]'
'`f` [*msglist*]'
: Lists the contents of '`From`' headers for a given set of messages.

'`z` [*arg*]' Presents message headers in pagefuls as described for `headers` command. When *arg* is '`.`', it is generally equivalent to `headers`. When *arg* is omitted or is '`+`', the next pageful of headers is displayed. If *arg* is '`-`', the previous pageful of headers is displayed. The latter two forms of `z` command may also take a numerical argument meaning the number of pages to skip before displaying the headers. For example:

```
& z +2
```

will skip two pages of messages before displaying the header summary.

'`size [`*msglist*`]`'
'`si [`*msglist*`]`'
Lists the message number and message size in bytes for each message in *msglist*.

'`folders`' Displays the value of `folder` variable.

'`summary`'
'`su`' Displays current mailbox summary. E.g.:

```
& summary
"/var/spool/mail/gray": 23 messages 22 unread
```

6.6.4.7 Displaying messages

'`print [`*msglist*`]`'
'`p [`*msglist*`]`'
'`type [`*msglist*`]`'
'`t [`*msglist*`]`'
Prints out the messages from *msglist*. The variable `crt` determines the minimum number of lines the body of the message must contain in order to be piped through pager command specified by environment variable `PAGER`. If `crt` is set to a numeric value, this value is taken as the minimum number of lines. Otherwise, if `crt` is set without a value then the height of the terminal screen is used to compute the threshold. The number of lines on screen is controlled by `screen` variable.

'`Print [`*msglist*`]`'
'`P [`*msglist*`]`'
'`Type [`*msglist*`]`'
'`T [`*msglist*`]`'
Like print but also prints out ignored header fields. For detailed description of ignored header lists, see Section 6.6.4.5 [Controlling Header Display], page 90.

'`decode [`*msglist*`]`'
'`dec [`*msglist*`]`'
Print a multipart message. The `decode` command decodes and prints out specified message parts. E.g.

```
& decode 15[2]
+--------------------------------------
| Message=15[2]
| Type=message/delivery-status
| encoding=7bit
+--------------------------------------
Content-Type: message/delivery-status
...
```

'`top` [*msglist*]'
'`to` [*msglist*]'
Prints the top few lines of each message in *msglist*. The number of lines printed is controlled by the variable `toplines` and defaults to five.

'`pipe` [*msglist*] [*shell-command*]'
'`|` [*msglist*] [*shell-command*]'
Pipe the contents of specified messages through *shell-command*. If *shell-command* is empty but the string variable `cmd` is set, the value of this variable is used as a command name.

6.6.4.8 Marking messages

'`tag` [*msglist*]'
'`ta` [*msglist*]'
Tag messages. The tagged messages can be referred to in message list using '`:t`' notation.

'`untag` [*msglist*]'
'`unt` [*msglist*]'
Clear tags from specified messages. To untag all messages tagged so far type

```
& untag :t
```

'`hold` [*msglist*]'
'`ho` [*msglist*]'
'`preserve` [*msglist*]'
'`pre` [*msglist*]'
Marks each message to be held in user's system mailbox. This command does not override the effect of `delete` command.

6.6.4.9 Disposing of messages

'`delete` [*msglist*]'
'`d` [*msglist*]'
Mark messages as deleted. Upon exiting with `quit` command these messages will be deleted from the mailbox. Until the end of current session the deleted messages can be referred to in message lists using :d notation.

'`undelete` [*msglist*]'
'`u` [*msglist*]'
Clear delete mark from the specified messages.

'`dp` [*msglist*]'
'`dt` [*msglist*]'
Deletes the current message and prints the next message. If *msglist* is specified, deletes all messages from the list and prints the message, immediately following last deleted one.

6.6.4.10 Saving messages

'`save [[`*msglist*`] `*file*`]`'
'`s [[`*msglist*`] `*file*`]`'
: Takes a message list and a file name and appends each message in turn to the end of the file. The name of file and number of characters appended to it is echoed on the terminal. Each saved message is marked for deletion as if with `delete` command, unless the variable `keepsave` is set.

'`Save [`*msglist*`]`'
'`S [`*msglist*`]`'
: Like `save`, but the file to append messages to is named after the sender of the first message in *msglist*. For example:

```
& from 14 15
 U  14 smith@noldor.org Fri Jun 30 18:11  14/358   The Save c
 U  15 gray@noldor.org  Fri Jun 30 18:30  8/245    Re: The Sa
& Save 14 15
"smith" 22/603
```

  i.e., 22 lines (603 characters) have been appended to the file "smith". If the file does not exist, it is created.

'`write [[`*msglist*`] `*file*`]`'
'`w [[`*msglist*`] `*file*`]`'
: Similar to `save`, except that only message body (without the header) is saved.

'`Write [`*msglist*`]`'
'`W [`*msglist*`]`'
: Similar to `Save`, except that only message body (without the header) is saved.

'`mbox [`*msglist*`]`'
'`mb [`*msglist*`]`'
'`touch [`*msglist*`]`'
'`tou [`*msglist*`]`'
: Mark list of messages to be saved in the user's mailbox ($HOME/mbox) upon exiting via `quit` command. This is the default action for all read messages, unless you have variable `hold` set.

'`copy [[`*msglist*`] `*file*`]`'
'`c [[`*msglist*`] `*file*`]`'
: Similar to `save`, except that saved messages are not marked for deletion.

'`Copy [`*msglist*`]`'
'`C [`*msglist*`]`'
: Similar to `Save`, except that saved messages are not marked for deletion.

6.6.4.11 Editing messages

These command allow to edit messages in a mailbox. *Please note*, that modified messages currently do not replace original ones. i.e. you have to save them explicitly using your editor's `save` command if you do not want the effects of your editing to be lost.

'`edit` [*msglist*]'
'`e` [*msglist*]'
: Edits each message in *msglist* with the editor, specified in `EDITOR` environment variable.

'`visual` [*msglist*]'
'`v` [*msglist*]'
: Edits each message in *msglist* with the editor, specified in `VISUAL` environment variable.

6.6.4.12 Scripting

Comments

The '`#`' character introduces an end-of-line comment. All characters until and including the end of line are ignored.

Displaying arbitrary text

The '`echo`' ('`ec`') command prints its arguments to stdout.

Sourcing external command files

The command '`source` *filename*' reads commands from the named file. Its minimal abbreviation is '`so`'.

Setting and unsetting the variables.

The mail variables may be set using '`set`' ('`se`') command. The command takes a list of assignments. The syntax of an assignment is

'*name*=*string*'
: Assign a string value to the variable. If *string* contains whitespace characters it must be enclosed in a pair of double-quote characters ('"')

'*name*=*number*'
: Assign a numeric value to the variable.

'*name*' Assign boolean `True` value.

'`no`*name*' Assign boolean `False` value.

Example:

```
& set askcc nocrt indentprefix="> "
```

This statement sets `askcc` to `True`, `crt` to `False`, and `indentprefix` to "`> `".

To unset mail variables use '`unset`'('`uns`') command. The command takes a list of variable names to unset.

Example: To undo the effect of the previous example, do:

```
& unset askcc crt indentprefix
```

Setting and unsetting shell environment variables.

Shell environment may be modified using '`setenv`' ('`sete`') command. The command takes a list of assignments. The syntax of an assignment is:

'*name*`=`*value*'
: If variable *name* does not already exist in the environment, then it is added to the environment with the value *value*. If *name* does exist, then its value in the environment is changed to *value*.

'*name*'
: Delete the variable *name* from the environment ("unset" it).

Conditional statements

The conditional statement allows to execute a set of mail commands depending on the mode the `mail` program is in. The conditional statement is:

```
if cond
...
else
...
endif
```

where '`...`' represents the set of commands to be executed in each branch of the statement. *cond* can be one of the following:

'`s`'
: True if `mail` is operating in mail sending mode.

'`r`'
: True if `mail` is operating in mail reading mode.

'`t`'
: True if stdout is a terminal device (as opposed to a regular file).

The conditional statements can be nested to arbitrary depth. The minimal abbreviations for '`if`', '`else`' and '`endif`' commands are '`i`', '`el`' and '`en`'.

Example:

```
if t
set crt prompt="& "
else
unset prompt
endif
if s
alt gray@farlep.net gray@mirddin.farlep.net
set
```

6.6.4.13 Aliasing

'`alias [alias [address...]]`'
'`a [alias [address...]]`'
'`group [alias [address...]]`'
'`g [alias [address...]]`'
: With no arguments, prints out all currently-defined aliases. With one argument, prints out that alias. With more than one argument, creates a new alias or changes an old one.

'`unalias [alias...]`'
'`una [alias...]`'
: Takes a list of names defined by alias commands and discards the remembered groups of users. The alias names no longer have any significance.

'`alternates name...`'
'`alt name...`'
: The alternates command is useful if you have accounts on several machines. It can be used to inform mail that the listed addresses are really you. When you reply to messages, mail will not send a copy of the message to any of the addresses listed on the alternates list. If the alternates command is given with no argument, the current set of alternate names is displayed.

6.6.4.14 Replying

'`mail [address...]`'
'`m [address...]`'
: Switches to compose mode. After composing the message, sends messages to the specified addresses.

'`reply [`*msglist*`]`'
'`respond [`*msglist*`]`'
'`r [`*msglist*`]`'
: For each message in *msglist*, switches to compose mode and sends the composed message to the sender and all recipients of the message.

'`Reply [`*msglist*`]`'
'`Respond [`*msglist*`]`'
'`R [`*msglist*`]`'
: Like `reply`, except that the composed message is sent only to originators of the specified messages.

'`followup [`*msglist*`]`'
'`fo [`*msglist*`]`'
: Switches to compose mode. After composing, sends the message to the originators and recipients of all messages in *msglist*.

'`Followup [`*msglist*`]`'
'`F [`*msglist*`]`'
: Similar to `followup`, but reply message is sent only to originators of messages in *msglist*.

6.6.4.15 Incorporating new mail

The `incorporate` (`inc`) command incorporates newly arrived messages to the displayed list of messages. This is done automatically before returning to `mail` command prompt if the variable `autoinc` is set.

6.6.4.16 Shell escapes

To run arbitrary shell command from `mail` command prompt, use `shell` (`sh`) command. If no arguments are specified, the command starts the user login shell. Otherwise, it uses its first argument as a file name to execute and all subsequent arguments are passed as positional parameters to this command. The `shell` command can also be spelled as `!`.

6.6.5 How to alter the behavior of mail

Following variables control the behavior of GNU `mail`:

`Sign`

Type: String.
Default: Unset.

Contains the filename holding users signature. The contents of this file is appended to the end of a message being composed by `~A` escape.

`appenddeadletter`

Type: Boolean.
Default: False.

If this variable is `True`, the contents of canceled letter is appended to the user's '`dead.letter`' file. Otherwise it overwrites its contents.

`askbcc`

Type: Boolean.
Default: False.

When set to `True` the user will be prompted to enter `Bcc` field before composing the message.

`askcc`

Type: Boolean.
Default: True.

When set to `True` the user will be prompted to enter `Cc` field before composing the message.

`asksub`

Type: Boolean.
Default: True in interactive mode, False otherwise.

When set to `True` the user will be prompted to enter `Subject` field before composing the message.

`autoinc`

Type: Boolean.
Default: True.

Automatically incorporate newly arrived messages.

`autoprint`

Type: Boolean.
Default: False.

Causes the delete command to behave like dp - thus, after deleting a message, the next one will be typed automatically.

`cmd`

Type: String.
Default: Unset.

Contains default shell command for `pipe`.

`columns`

Type: Numeric.
Default: Detected at startup by querying the terminal device. If this fails, the value of environment variable `COLUMNS` is used.

This variable contains the number of columns on terminal screen.

`crt`

Type: Boolean or Numeric
Default: True in interactive mode, False otherwise.

The variable `crt` determines the minimum number of lines the body of the message must contain in order to be piped through pager command specified by environment variable `PAGER`. If `crt` is set to a numeric value, this value is taken as the threshold. Otherwise, if `crt` is set without a value, then the height of the terminal screen is used to compute the threshold. The number of lines on screen is controlled by `screen` variable.

`dot`

Type: Boolean.
Default: False.

If `True`, causes `mail` to interpret a period alone on a line as the terminator of a message you are sending.

`editheaders`

Type: Boolean.
Default: False.

When set, `mail` will include message headers in the text to be the `~e` and `~v` escapes, thus allowing you to customize the headers.

`escape`

Type: String.
Default: `~`

If defined, the first character of this option gives the character to denoting escapes.

`folder`

Type: String.
Default: Unset.

The name of the directory to use for storing folders of messages. If unset, $HOME is assumed.

`header`

Type: Boolean.
Default: True, unless started with '`--nosum`' ('`-N`') option.

Whether to run `headers` command automatically after entering interactive mode.

`hold`

Type: Boolean.
Default: False.

When set to `True`, the read or saved messages will be stored in user's mailbox ('`$HOME/mbox`'). Otherwise, they will be held in system mailbox also. This option is in effect only when operating upon user's system mailbox.

`ignore`

Type: Boolean.
Default: False.

When set to `True`, `mail` will ignore keyboard interrupts when composing messages. Otherwise an interrupt will be taken as a signal to abort composing.

`ignoreeof`

Type: Boolean.
Default: False.

Controls whether typing EOF character terminates the letter being composed.

`indentprefix`

Type: String.
Default: `"\t"` (a tab character).

String used by the `~m` tilde escape for indenting quoted messages.

`keepsave`

Type: Boolean.
Default: False.

Controls whether saved messages should be kept in system mailbox too. This variable is in effect only when operating upon a user's system mailbox.

`metoo`

Type: Boolean.
Default: False.

Usually, when an alias is expanded that contains the sender, the sender is removed from the expansion. Setting this option causes the sender to be included in the group.

`mode`

Type: String.
Default: The name of current operation mode.

Setting this variable does not affect the operation mode of the program.

`noregex`

Type: Boolean.
Default: False.

Setting this to `True` enables use of regular expressions in '`/.../`' message specifications.

`outfolder`

Type: String.
Default: Unset.

Contains the directory in which files created by `save`, `write`, etc. commands will be stored. When unset, current directory is assumed.

`page`

Type: Boolean.
Default: False.

If set to `True`, the `pipe` command will emit a linefeed character after printing each message.

`prompt`

Type: String.
Default: `"? "`

Contains the command prompt sequence.

`quiet`

Type: Boolean.
Default: False.

When set, suppresses the output of the startup banner.

`quit`

Type: Boolean.
Default: False, unless started with '`--quit`' ('`-q`') option.

When set, causes keyboard interrupts to terminate the program.

`rc`

Type: Boolean.
Default: True, unless started with '`--norc`' ('`-N`') option.

When this variable is set, `mail` will read the system-wide configuration file upon startup. See Section 6.6.6 [Mail Configuration Files], page 101.

`record`

Type: String.
Default: Unset.

When set, any outgoing message will be saved to the named file.

`save`
: Type: Boolean.
Default: True.

When set, the aborted messages will be stored in the user's '`dead.file`'. See also `appenddeadletter`.

`screen`
: Type: Numeric.
Default: Detected at startup by querying the terminal device. If this fails, the value of environment variable `LINES` is used.

This variable contains the number of lines on terminal screen.

`sendmail`
: Type: String.
Default: sendmail:/usr/lib/sendmail

Contains the URL of mail transport agent.

`sign`
: Type: String.
Default: Unset.

Contains the user's signature. The contents of this variable is appended to the end of a message being composed by `~a` escape. Use `Sign` variable, if your signature occupies more than one line.

`subject`
: Type: String.
Default: Unset.

Contains default subject line. This will be used when `asksub` is off.

`toplines`
: Type: Numeric.
Default: 5

Number of lines to be displayed by `top` and `Top` commands.

`verbose`
: Type: Boolean.
Default: False.

When set, the actual delivery of messages is displayed on the user's terminal.

6.6.6 Personal and system-wide configuration files

Upon startup, `mail` reads the contents of the two command files: the system-wide configuration file, and the user's configuration file. Each line read from these files is processed like a usual `mail` command.

When run with '`--norc`' ('`-N`') option, `mail` does not read the contents of system-wide configuration file. The user's file, if it exists, is always processed.

The user's configuration file is located in the user's home directory and is named '`.mailrc`'. The location and name of the system-wide configuration file is determined when configuring the package via '`--with-mail-rc`' option. It defaults to '*sysconfdir*`/mail.rc`'.

6.7 mail.local — Deliver mail to the local mailbox.

`mail.local` reads the standard input up to an end-of-file and appends the received data to the local mailboxes.

6.7.1 Invoking mail.local

The program uses following option groups: See Section 6.2.2 [mailbox], page 75, See Section 6.2.6 [auth], page 76, See Section 6.2.7 [logging], page 77, See Section 6.10 [sieve], page 107.

'`-f` *addr*'
'`--from` *addr*'
: Specify the sender's name. This option forces `mail.local` to add '`From `' envelope to the beginning of the message. If it is not specified, `mail.local` first looks into the first line from the standard input. If it starts with '`From `', it is assumed to contain a valid envelope. If it does not, `mail.local` creates the envelope by using current user name and date.

'`-h`'
'`--help`'
: Display this help and exit.

'`-L`'
'`--license`'
: Display GNU General Public License and exit.

'`-m` *path*'
'`--mail-spool` *path*'
: Specify path to mailspool directory.

'`-q`'
'`--quota-db` *file*'
: Specify path to mailbox quota database (see Section 6.7.3 [Mailbox Quotas], page 104).

'`-s` *pattern*'
'`--source` *pattern*'
: Set name pattern for user-defined mail filters written in Scheme (see Section 6.7.5 [Scheme Filters], page 104). The metacharacters '`%u`' and '`%h`' in the pattern are expanded to the current recipient user name and home directory correspondingly.

  This option is available only if the package has been configured to use Guile extension language.

'`-S` *pattern*'
'`--sieve` *pattern*'
: Set name pattern for user-defined mail filters written is Sieve (see Section 6.7.4 [Sieve Filters], page 104). The metacharacters '`%u`' and '`%h`' in the pattern are expanded to the current recipient user name and home directory correspondingly.

'`-t` *number*'
'`--timeout` *number*'
: Wait *number* seconds for acquiring the lockfile. If it doesn't become available after that amount of time, return failure. The timeout defaults to 5 minutes.

'`-x` *flags*'
'`--debug` *flags*'
: Enable debugging. The debugging information will be output using syslog. The *flags* is a string consisting of the following flags: Debug flags are:

  '`g`' Start with guile debugging evaluator and backtraces. This is convenient for debugging user-defined filters (see Section 6.7.5 [Scheme Filters], page 104).

  '`T`' Enable libmailutil traces (MU_DEBUG_TRACE).

  '`P`' Enable network protocol traces (MU_DEBUG_PROT)

  '`t`' Enable sieve trace (MU_SIEVE_DEBUG_TRACE)

  '`l`' Enable sieve action logs

  The digits in the range '`0`' – '`9`' used in *flags* set `mail.local` debugging level.

'`-v`'
'`--version`'
: Display program version and exit.

'`--ex-multiple-delivery-success`'
: Don't return errors when delivering to multiple recipients.

'`--ex-quota-tempfail`'
: Return temporary failure if disk or mailbox quota is exceeded. By default, 'service unavailable' is returned if the message exceeds the mailbox quota.

6.7.2 Using mail.local with various MTAs

This section explains how to invoke `mail.local` from configuration files of various Mail Transport Agents.

All examples in this section suppose that `mail.local` must receive following command line switches:

```
-s %h/.filter.scm -q /etc/mail/userquota
```

6.7.2.1 Using mail.local with sendmail.

The `mail.local` must be invoked from the local mailer definition in the '`sendmail.cf`' file. It must have the following flags set '`lswS`', meaning the mailer is local, the quote characters should be stripped off the address before invoking the mailer, the user must have a valid account on this machine and the userid should not be reset before calling the mailer. Additionally, '`fn`' flags may be specified to allow `mail.local` to generate usual '`From `' envelope instead of the one supplied by sendmail.

Here is an example of mailer definition in '`sendmail.cf`'

```
Mlocal, P=/usr/local/libexec/mail.local,
        F=lsDFMAw5:/|@qSPfhn9,
        S=EnvFromL/HdrFromL, R=EnvToL/HdrToL,
        T=DNS/RFC822/X-Unix,
        A=mail -s %h/.filter.scm -q /etc/mail/userquota $u
```

To define local mailer in 'mc' source file, it will suffice to set:

```
define('LOCAL_MAILER_PATH', '/usr/local/libexec/mail.local')
define('LOCAL_MAILER_ARGS',
       'mail -s %h/.filter.scm -q /etc/mail/userquota $u')
```

6.7.2.2 Using mail.local with exim.

Using `mail.local` with exim is quite straightforward. The following example illustrates the definition of appropriate transport and director in '`exim.conf`':

```
# transport
mail_local_pipe:
  driver = pipe
  command = /usr/local/libexec/mail.local -s %h/.filter.scm \
              -q /etc/mail/userquota $local_part
  return_path_add
  delivery_date_add
  envelope_to_add

# director
mail_local:
  driver = localuser
  transport = mail_local_pipe
```

6.7.3 Setting up mailbox quotas

6.7.4 Implementing user-defined Sieve mail filters.

6.7.5 Implementing user-defined Scheme mail filters.

6.8 messages — Count the number of messages in a mailbox.

`Messages` prints on standard output the number of messages contained in each folder specified in command line. If no folders are specified, it operates upon user's system mailbox. For each folder, the following output line is produced:

`Number of messages in` *folder*`:` *number*

where *folder* represents the folder name, *number* represents the number of messages.

The program uses following option groups: See Section 6.2.2 [mailbox], page 75.

The program accepts following command line options:

'`-q`'
'`--quite`'
'`-s`'
'`--silent`'
: Be quiet. Display only number of messages per mailbox, without leading text.

'`-?`'
'`--help`'
: Output help message and exit.

'`--usage`'
: Output short usage summary and exit.

'`-V`'
'`--version`'
: Output program version and exit.

6.9 readmsg — Extract messages from a folder.

The program, readmsg, extracts with the selection argument messages from a mailbox. Selection can be specify by:

1. A lone "*" means select all messages in the mailbox.
2. A list of message numbers may be specified. Values of "0" and "$" in the list both mean the last message in the mailbox. For example:

   ```
   readmsg 1 3 0
   ```

   extracts three messages from the folder: the first, the third, and the last.
3. Finally, the selection may be some text to match. This will select a mail message which exactly matches the specified text. For example,

   ```
   readmsg staff meeting
   ```

   extracts the message which contains the words "staff meeting." Note that it will not match a message containing "Staff Meeting" - the matching is case sensitive. Normally only the first message which matches the pattern will be printed.

Command line options

'`-a`'
'`--show-all`'
: If a pattern is use for selection show all messages that match pattern by default only the first one is presented.

'`-d`'
'`--debug`'
: Display mailbox debugging information.

'`-f` *MAILBOX*'
'`--folder=`*MAILBOX*'
: Specified the default mailbox.

'`-h`'
'`--header`'
: Show the entire header and ignore the weedlist.

'`-n`'
'`--no-header`'
: Do not print the message header.

'`-p`'
'`--form-feed`'
: Put form-feed (Control-L) between messages instead of newline.

'`-w` *weedlist*'
'`--weedlist=`*weedlist*'
: A whitespace or coma separated list of header names to show per message. Default is –weedlist="From Subject Date To CC Apparently-"

6.10 sieve

Sieve is a language for filtering e-mail messages at time of final delivery, described in RFC 3028. GNU Mailutils provides two implementations of this language: a stand-alone *sieve interpreter* and a *sieve translator and filter*. The following sections describe these utilities in detail.

6.10.1 A Sieve Interpreter

Sieve interpreter `sieve` allows to apply Sieve scripts to an arbitrary number of mailboxes. GNU `sieve` implements a superset of the Sieve language as described in RFC 3028. See Section 5.2 [Input Language], page 54, for a description of the Sieve language. See Section 5.3 [Extensions], page 67, for a discussion of differences between the GNU implementation of Sieve and its standard.

6.10.1.1 Invocation

The `sieve` invocation syntax is:

```
sieve [options] script
```

where *script* denotes the filename of the sieve program to parse, and *options* is one or more of the following:

'`-c`'
'`--compile-only`'
: Compile script and exit.

'`-d[`*flags*`]`'
'`--debug[=`*flags*`]`'
: Specify debug flags. The *flags* argument is a sequence of one or more of the following letters:

'`g`'	Enable main parser traces
'`T`'	Enable mailutil traces
'`P`'	Trace network protocols
'`t`'	Enable sieve trace
'`i`'	Trace the program instructions

See Section 6.10.1.2 [Logging and Debugging], page 108, for detailed discussion of these.

'`-D`'
'`--dump`'
: Compile the script, dump disassembled code on standard output and exit.

'`-e` *address*'
'`--email` *address*'
: Override the user email address. This is useful for `reject` and `redirect` actions. By default, the user email address is deduced from the user name and the full name of the machine where sieve is executed.

'`-f`'
'`--mbox-url=`*mbox*'
: Mailbox to sieve (defaults to user's system mailbox)

'`-k`'
'`--keep-going`'
: Keep on going if execution fails on a message

'`-n`'
'`--no-actions`'
: Dry run: do not execute any actions, just print what would be done.

'`-t` *ticket*'
'`--ticket=`*ticket*'
: Ticket file for mailbox authentication

'`-v`'
'`--verbose`'
: Log all actions executed.

Apart from these, `sieve` understands the options from the following groups: `sieve`, `mailbox`, `mailer`, `logging`.

6.10.1.2 Logging and debugging

The default behavior of `sieve` is to remain silent about anything except errors. However, it is sometimes necessary to see which actions are executed and on which messages. This is particularly useful when debugging the sieve scripts. The '`--verbose`' ('`-v`') option outputs log of every action executed.

Option '`--debug`' allows to produce even more detailed debugging information. This option takes an argument specifying the debugging level to be enabled. The argument can consist of the following letters:

``t''
: This flag enables sieve tracing. It means that every test will be logged when executed.

``T''
: This flag enables debugging of underlying `mailutils` library.

``P''
: Trace network protocols: produces log of network transactions executed while running the script.

``g''
: Enable main parser traces. This is useful for debugging the sieve grammar.

``i''
: Trace the program instructions. It is the most extensive debugging level. It produces the full execution log of a sieve program, showing each instruction and states of the sieve machine. It is only useful for debugging the code generator.

Note, that there should be no whitespace between the short variant of the option ('`-d`'), and its argument. Similarly, when using long option ('`--debug`'), its argument must be preceded by equal sign.

If the argument to '`--debug`' is omitted, it defaults to '`TPt`'.

Option '`--dump`' produces the disassembled dump of the compiled sieve program.

By default `sieve` output all diagnostics on standard error and verbose logs on standard output. This behaviour is changed when '`--log-facility`' is given in the command line (see Section 6.2.7 [logging], page 77). This option causes `sieve` to output its diagnostics to the given syslog facility.

6.10.1.3 Extending Sieve

The basic set of sieve actions, tests and comparators may be extended using loadable extensions. Usual `require` mechanism is used for that.

When processing arguments for `require` statement, `sieve` uses the following algorithm:

1. Look up the name in a symbol table. If the name begins with '`comparator-`' it is looked up in the comparator table. If it begins with '`test-`', the test table is used instead. Otherwise the name is looked up in the action table.
2. If the name is found, the search is terminated.
3. Otherwise, transform the name. First, any '`comparator-`' or '`test-`' prefix is stripped. Then, any character other than alphanumeric characters, '`.`' and '`,`' is replaced with dash ('`-`'). The name thus obtained is used as a file name of an external loadable module.
4. Try to load the module. The module is searched in the following search paths (in the order given):
   1. Mailutils module directory. By default it is '`$prefix/lib/mailutils`'.
   2. The value of the environment variable LTDL_LIBRARY_PATH.
   3. Additional search directories specified with the `#searchpath` directive.
   4. System library search path: The system dependent library search path (e.g. on Linux it is set by the contents of the file '`/etc/ld.so.conf`' and the value of the environment variable LD_LIBRARY_PATH).

   The value of LTDL_LIBRARY_PATH and LD_LIBRARY_PATH must be a colon-separated list of absolute directories, for example, '`"/usr/lib/mypkg:/lib/foo"`'.

   In any of these directories, `sieve` first attempts to find and load the given filename. If this fails, it tries to append the following suffixes to the file name:
   1. the libtool archive extension '`.la`'
   2. the extension used for native dynamic libraries on the host platform, e.g., '`.so`', '`.sl`', etc.
5. If the module is found, `sieve` executes its initialization function (see below) and again looks up the name in the symbol table. If found, search terminates successfully.
6. If either the module is not found, or the symbol wasn't found after execution of the module initialization function, search is terminated with an error status. `sieve` then displays the following diagnostic message:

```
source for the required action NAME is not available
```

6.10.2 A Sieve to Scheme Translator and Filter

A Sieve to Scheme Translator `sieve.scm` translates a given Sieve script into an equivalent Scheme program and optionally executes it. The program itself is written in Scheme and requires presence of Guile 1.4 on the system. For more information on Guile refer to section "Overview" in *The Guile Reference Manual*.

'`-f` *filename*'
'`--file` *filename*'
: Set input file name.

'`-o` *filename*'
'`--output` *filename*'
: Set output file name

'`-L` *dirname*'
'`--lib-dir` *dirname*'
: Set sieve library directory name

'`-d` *level*'

'`--debug` *level*'
: Set debugging level

The Scheme programs produced by `sieve.scm` can be used with `guimb` or `mail.local`.

6.11 guimb — A mailbox scanning and processing language.

`Guimb` is for mailboxes what `awk` is for text files. It processes mailboxes, applying the user-supplied scheme procedures to each of them in turn and saves the resulting output in mailbox format.

The program uses following option groups: See Section 6.2.2 [mailbox], page 75.

6.11.1 Specifying Scheme Program to Execute

The Scheme program or expression to be executed is passed to `guimb` via the following options:

'`-s` *file*'
'`--source` *file*'
: Load Scheme source code from *file*.

'`-c` *expr*'
'`--code` *expr*'
: Execute given scheme expression.

The above switches stop further argument processing, and pass all remaining arguments as the value of `(command-line)`.

If the remaining arguments must be processed by `guimb` itself, use following options:

'`-e` *expr*'
'`--expression` *expr*'
: Execute scheme expression.

'`-f` *file*'
'`--file` *file*'
: Load Scheme source code from *file*.

You can specify both of them. In this case, the *file* is read first, then *expr* is executed. You may still pass any additional arguments to the script using '`--guile-arg`' option (see Section 6.11.3 [Passing Options to Scheme], page 112).

6.11.2 Specifying Mailboxes to Operate Upon

There are four basic ways of passing mailboxes to `guimb`.

`guimb [`*options*`] [`*mailbox*`...]`
: The resulting mailbox is not saved, unless the user-supplied scheme program saves it.

`guimb [`*options*`] --mailbox` *defmbox*
: The contents of *defmbox* is processed and is replaced with the resulting mailbox contents. Useful for applying filters to user's mailbox.

`guimb [`*options*`] --mailbox` *defmbox* `mailbox [mailbox...]`
: The contents of specified mailboxes is processed, and the resulting mailbox contents is appended to *defmbox*.

`guimb` [*options*] `--user` *username* [`mailbox...`]
: The contents of specified mailboxes is processed, and the resulting mailbox contents is appended to the user's system mailbox. This allows to use `guimb` as a mail delivery agent.

If no mailboxes are specified in the command line, `guimb` reads and processes the system mailbox of the current user.

6.11.3 Passing Options to Scheme

Sometimes it is necessary to pass some command line options to the scheme procedure. There are three ways of doing so.

When using '`--source`' ('`-s`') or '`--code`' ('`-c`') options, all the rest of the command line following the option's argument is passed to Scheme program verbatim. This allows for making guimb scripts executable by the shell. If your system supports '`#!`' magic at the start of scripts, add the following two lines to the beginning of your script to allow for its immediate execution:

```
#! /usr/local/bin/guimb -s
!#
```

(replace '`/usr/local/bin/`' with the actual path to the `guimb`).

Otherwise, if you use '`--file`' or '`--expression`' options, the additional arguments may be passed to the Scheme program '`-g`' ('`--guile-arg`') command line option. For example:

```
guimb --guile-arg -opt --guile-arg 24 --file progfile
```

In this example, the scheme procedure will see the following command line:

```
progfile -opt 24
```

Finally, if there are many arguments to be passed to Scheme, it is more convenient to enclose them in '`-{`' and '`-}`' escapes:

```
guimb -{ -opt 24 -} --file progfile
```

6.11.4 Guimb Invocation Summary

This is a short summary of the command line options available to `guimb`.

'`-d`'
'`--debug`' Start with debugging evaluator and backtraces.

'`-e EXPR`'
'`--expression EXPR`'
: Execute given Scheme expression.

'`-m` *path*'
'`--mail-spool=`*path*'
: Set path to the mailspool directory

'`-f PROGFILE`'
'`--file PROGFILE`'
: Read Scheme program from PROGFILE.

‘`-g ARG`’
‘`--guile-command ARG`’
: Append ARG to the command line passed to Scheme program.

‘`-{ ... -}`’
: Pass all command line options enclosed between ‘`-{`’ and ‘`-}`’ to Scheme program.

‘`-m`’
‘`--mailbox MBOX`’
: Set default mailbox name.

‘`-u`’
‘`--user NAME`’
: Act as local MDA for user NAME.

‘`-h`’
‘`--help`’ Display help message.

‘`-v`’
‘`--version`’
: Display program version.

6.12 Comsat daemon

Comsatd is the server which receives reports of incoming mail and notifies users, wishing to get this service. It can be started either from '`inetd.conf`' or as a standalone daemon.

6.12.1 Starting comsatd

`Comsatd` uses following option groups: See Section 6.2.2 [mailbox], page 75, See Section 6.2.5 [daemon], page 76, See Section 6.2.7 [logging], page 77.

'`-c` *file*'
'`--config` *file*'
: Read configuration from given *file*. For more information about comsatd configuration files, see Section 6.12.2 [Configuring comsatd], page 114.

'`-d`'
'`--daemon`'
: Run as a standalone daemon.

'`-i`'
'`--inetd`'
: The server is started from '`/etc/inetd.conf`' file:

```
comsat dgram  udp wait  root  /usr/sbin/comsatd \
comsatd -c /etc/comsat.conf
```

This is the default operation mode.

'`-m` *path*'
'`--mail-spool=`*path*'
: Set path to the mailspool directory

'`-p` *number*'
'`--port` *number*'
: Specify the port number to listen on. Default is 512.

'`-v`'
'`--version`'
: Output version and exit successfully.

'`-h`'
'`--help`'
: Display short help message and exit.

6.12.2 Configuring comsatd

The configuration parameters for comsatd are kept in a single configuration file. The file uses line-oriented format: each line contains a single statement. Comments are introduced with the '`#`' sign and empty lines are ignored. You can specify the configuration file to use by using '`-c`' or '`--config`' command line switch.

The configuration file statements can logically be subdivided into *General Settings*, *Security Settings* and *Access Control Lists*. The following sections address each of these statement group in detail.

6.12.2.1 General Settings

These statements control the general behavior of the comsat daemon:

max-lines *number*
: Set maximum number of message body lines to be output.

allow-biffrc (yes | no)
: Enable or disable processing of user's '`.biffrc`' file. By default, it is enabled.

6.12.2.2 Security Settings

These statements control the way `comsatd` fights possible flooding attacks.

max-requests *number*
: Set maximum number of incoming requests per '`request-control-interval`'.

request-control-interval *number*
: Set the request control interval (seconds).

overflow-delay-time *number*
: Set the initial amount of time to sleep, after the first overflow occurs.

overflow-control-interval *number*
: Set the overflow control interval. If two consecutive overflows happen within *number* seconds, the overflow-delay-time is doubled.

6.12.2.3 Access Control Lists

Access control lists determine from which addresses `comsatd` will receive mail notification messages.

The access control lists are introduced in configuration file using keyword '`acl`'. General format for an ACL rule is

```
acl action netlist
```

Here, *action* specifies the action to be taken when a request arrives from one of the networks, listed in *netlist*. There are two possible actions: '`allow`' and '`deny`'.

The *netlist* is a whitespace-separated list of network numbers. Each network number may be specified in one of the following forms:

netnum
: Means a single host with IP address *netnum*.

netnum/*netmask*
netnum/*masklen*
'`any`'
: Denotes any IP address. It is equivalent to '`0.0.0.0/0`'.

Upon receiving a notification message, `comsatd` compares its source address against each ACL rule in the order of their appearance in the configuration file. The first rule that matches the packet determines whether the message will be processed or rejected. If no matching rule was found, the default rule applies. Currently, default rule is

```
acl allow any
```

If you don't need such behavior, specify the default rule explicitly. For example, the common use would be:

```
acl allow 127.0.0.1
acl deny any
```

which makes `comsatd` receive the notification messages from localhost only.

6.12.3 A per-user configuration file.

By default, when a notification arrives, `comsatd` prints subject, from headers and the first five lines from the new message to the user's tty. The user is allowed to change this behavior by using his own configuration file. This file should be located in the user's home directory and should be named '`.biffrc`'. It must be owned by the user and have its permissions bits set to 0600. (*Please note*, that the use of per-user configuration files may be disabled, by specifying '`allow-biffrc no`' in the main configuration file, see see Section 6.12.2 [Configuring comsatd], page 114).

The '`.biffrc`' file consists of a series of statements. Each statement occupies one line and defines an action to be taken upon arrival of a new mail. Very long lines may be split using '\' as the last character on the line. As usual, comments may be introduced with '`#`' character.

The actions specified in '`.biffrc`' file are executed in turn. The following actions are defined:

beep Produce an audible signal.

echo *string*
: Output *string* to user's terminal device.

exec *prog arglist*
: Execute program *prog* with arguments from *arglist*. *prog* must be specified with absolute pathname. It may not be a setuid or setgid program.

In the description above, *string* denotes any sequence of characters. This sequence must be enclosed in a pair of double-quotes, if it contains whitespace characters. The '\' character inside a string starts a C escape sequence. Following meta-characters may be used in strings:

$u Expands to username

$h Expands to hostname

$H{name}
: Expands to value of message header '`name`'.

$B(*c*,*l*) Expands to message body. *c* and *l* give maximum number of characters and lines in the expansion. When omitted, they default to 400, 5.

6.12.3.1 Example I.

Dump to the user's terminal the contents of '`From`' and '`Subject`' headers followed by at most 5 lines of message body.

```
echo "Mail to \a$u@$h\a\n---\n\
From: $H{from}\n\
Subject: $H{Subject}\n\
---\n\
$B(,5)\
---\n"
```

6.12.3.2 Example II.

Produce a bell, then pop up the xmessage window on display :0.0 with the text formatted in the same manner as in the previous example.

```
beep
exec /usr/X11R6/bin/xmessage \
-display :0.0 -timeout 10 "Mail to $u@$h \n---\n\
From: $H{from}\n\
Subject: $H{Subject}\n\
---\n\
$B(,5)\
---\n"
```

6.13 mailutils-config — Get the information about the Mailutils build.

This program is designed for developers wishing to link their programs against libmailbox. It allows to examine the particulars of the current build of Mailutils and to get the command line parameters necessary for compiling and linking an application with Mailutils libraries.

6.13.1 Getting Compiler Flags.

When invoked with the option '`--compile`', or its short form '`-c`', `mailutils-config` prints the flags that must be given to the compiler for compiling the program using Mailutils functions. An example usage:

```
cc -omyprog.o `mailutils-config --compile` myprog.c
```

6.13.2 Getting Loader Flags

The '`--link`', or its short form '`-l`' prints to the standard output the loader flags necessary to link a program against Mailutils libraries.

When invoked without arguments, it produces the flags necessary to link against the basic library of Mailutils: '`libmailbox`'. Up to two arguments may be given that alter this behavior. These are:

'`auth`' Print flags to link against '`libmuauth`', the library adding new authentication methods to '`libmailbox`'.

'`guile`' Print flags to link against '`libmu_scm`', the Guile interface library.

Both arguments may be given simultaneously, e.g.:

```
cc -omyprog myprog.o `mailutils-config --link auth guile`
```

6.13.3 Obtaining General Build Information

The '`--info`', or '`-i`' retrieves the options (flags) used when building Mailutils. It may be used with or without arguments.

When used without arguments, it prints the list of all build flags, e.g.:

```
$ mailutils-config --info
VERSION=0.1.3
USE_LIBPAM
HAVE_LIBLTDL
WITH_GUILE
WITH_READLINE
HAVE_MYSQL
ENABLE_VIRTUAL_DOMAINS
ENABLE_IMAP
ENABLE_POP
ENABLE_MH
ENABLE_SMTP
ENABLE_SENDMAIL
```

This option also accepts any number of arguments. When these are given, each argument is treated as a name of a build flag. `Mailutils-config` checks if such a flag was defined and prints its full name if so. It exits with zero code if all the flags given on the command line are defined. Otherwise, it exits with code of 1.

The comparison of the flag names is case-insensitive. The arguments given need not include the leading prefix (i.e. the characters up to and including the first underscore character).

Given the previous example, the invocation

```
$ mailutils --info readline use_libpam pop
```

will produce the following output:

```
WITH_READLINE
USE_LIBPAM
ENABLE_POP
```

and will exit with a zero status.

The following command:

```
$ mailutils --info readline gssapi pop
```

will exit with status 1, and will print:

```
WITH_READLINE
ENABLE_POP
```

since `WITH_GSSAPI` flag is not defined.

The flags and their meanings are:

USE_LIBPAM
: The Mailutils uses PAM libraries.

HAVE_LIBLTDL
: The GNU wrapper library '`libltdl`' is present and is used by Mailutils. See section "Using libltdl" in *Using libltdl*, for more information on '`libltdl`' library.

WITH_BDB2
: Support for Berkeley DB is compiled in (the package was configured with '`--with-db2`' option).

WITH_NDBM
: Support for NDBM is compiled in (the package was configured with '`--with-ndbm`' option).

WITH_OLD_DBM
: Support for old style DBM is compiled in (the package was configured with '`--with-dbm`' option).

WITH_GDBM
: Support for GDBM is compiled in (the package was configured with '`--with-gdbm`' option). See section "Introduction" in *GNU DBM Manual*, for information about this library.

WITH_GSSAPI
: Support for GSSAPI is compiled in (the package was configured with '`--with-gssapi`' option).

WITH_GUILE
: Support for Guile extension language is built (the package was configured with '`--with-guile`' option). See section "Overview" in *The Guile Reference Manual*, for information about Guile.

WITH_PTHREAD
: The POSIX thread support is compiled in.

WITH_READLINE
: The readline support is enabled (the package was configured with '`--with-readline`' option). See section "Top" in *GNU Readline Library*, for more information.

HAVE_MYSQL
: Authentication via MySQL is supported (the package was configured with '`--enable-mysql`' option).

ENABLE_VIRTUAL_DOMAINS
: Support for mail virtual domains is enabled (the package was configured with '`--enable-virtual-domains`' option).

ENABLE_IMAP
: Support for IMAP4 protocol is enabled.

ENABLE_POP
: Support for POP3 protocol is enabled.

ENABLE_MH
: Support for mailboxes in MH format is enabled.

ENABLE_SMTP
: Support for SMTP mailer is enabled.

ENABLE_SENDMAIL
: Support for Sendmail mailer is enabled.

7 Reporting Bugs

Email bug reports to `bug-mailutils@gnu.org`. Be sure to include the word "mailutils" somewhere in the "Subject:" field.

As the purpose of bug reporting is to improve software, please be sure to include maximum information when reporting a bug. The information needed is:

Version of the package you are using.

Compilation options used when configuring the package.

Conditions under which the bug appears.

The archives of bug-mailutils mailing list are available from `http://mail.gnu.org/mailman/listinfo/bug-mailutils`.

8 Getting news about GNU mailutils

The two places to look for any news regarding GNU Mailutils are the Mailutils homepage at `http://www.gnu.org/software/mailutils`, and the project page at `http://savannah.gnu.org/projects/mailutils`.

The updated versions of this manual are available online from `http://www.gnu.org/software/mailutils/`

9 Acknowledgement

In no particular order,

- Jakob Kaivo `jkaivo@ndn.net`,
- Jeff Bailey `jbailey@gnu.org`,
- Sean Perry `shaleh@debian.org`,
- Thomas Fletcher `thomasf@qnx.com`,
- Dave Inglis `dinglis@qnx.com`,
- Brian Edmond `briane@qnx.com`,
- Sam Roberts `sroberts@uniserve.com`,
- Sergey Poznyakoff `gray@Mirddin.farlep.net`,
- François Pinard `pinard@IRO.UMontreal.CA`.
- Jordi Mallach `jordi@sindominio.net`

Function Index

This is an alphabetical list of all Mailutils functions.

*

*sieve_get_data 50

A

address 62
address_aget_email 26
address_create 25
address_createv 26
address_destroy 26
address_get_comments 27
address_get_count 28
address_get_domain 27
address_get_email 26, 27
address_get_local_part 27
address_get_personal 26
address_get_route 28
address_is_group 28
address_t 25
address_to_string 28
attribute_copy 19
attribute_create 18
attribute_destroy 18
attribute_is_answered 18
attribute_is_deleted 18
attribute_is_draft 18
attribute_is_equal 19
attribute_is_flagged 18
attribute_is_read 18
attribute_is_recent 18
attribute_is_seen 18
attribute_set_answered 18
attribute_set_deleted 18
attribute_set_draft 18
attribute_set_flagged 18
attribute_set_read 18
attribute_set_recent 18
attribute_set_seen 18
attribute_to_string 19
attribute_unset_answered 19
attribute_unset_deleted 19
attribute_unset_draft 19
attribute_unset_flagged 19
attribute_unset_read 19
attribute_unset_recent 19
attribute_unset_seen 18
auth_authenticate 24
auth_create 24
auth_destroy 24
auth_epilogue 24
auth_prologue 24

B

body_create 17
body_destroy 17
body_get_filename 18
body_get_stream 17
body_lines 18
body_set_filename 18
body_set_stream 18
body_size 18

C

char* 33

D

decoder_stream_create 20
discard 65

E

encoder_stream_create 20
envelope 63
envelope_create 15
envelope_date 15
envelope_destroy 15
envelope_get_message 15
envelope_sender 15
envelope_set_date 15
envelope_set_sender 15
exists 63

F

false 62
file_stream_create 19
fileinto 65
folder_close 5
folder_create 5
folder_delete 5
folder_destroy 5
folder_get_authority 5
folder_get_debug 5
folder_get_observable 5
folder_get_stream 5
folder_get_url 6
folder_list 5
folder_lsub 5
folder_open 5
folder_rename 5
folder_set_authority 6
folder_set_debug 5

folder_set_stream 5
folder_set_url 6
folder_subscribe 5
folder_unsubscribe 5

H

header 64
header_aget_value 17
header_create 15
header_destroy 15
header_get_stream 17
header_get_value 17
header_set_lines 17
header_set_size 17
header_set_value 16

I

iterator_create 23
iterator_current 23
iterator_destroy 23
iterator_first 23
iterator_is_done 23
iterator_next 23

K

keep 65

L

locker_create 31
locker_destroy 31
locker_lock 31
locker_touchlock 31
locker_unlock 31

M

mailbox_append_message 6
mailbox_close 6
mailbox_create 7
mailbox_create_default 7
mailbox_destroy 7
mailbox_expunge 7
mailbox_get_debug 7
mailbox_get_folder 7
mailbox_get_locker 7
mailbox_get_message 8
mailbox_get_observable 8
mailbox_get_property 8
mailbox_get_size 8
mailbox_get_stream 8
mailbox_get_ticket 8
mailbox_get_url 9
mailbox_is_modified 9
mailbox_message_unseen 9
mailbox_messages_count 9
mailbox_messages_recent 9
mailbox_open 9
mailbox_scan 10
mailbox_set_locker 10
mailbox_set_stream 10
mailbox_set_ticket 10
mailbox_t 6
mailbox_uidnext 10
mailbox_uidvalidity 10
mailer_close 11
mailer_create 11
mailer_destroy 11
mailer_get_debug 11
mailer_get_observable 11
mailer_get_property 11
mailer_get_stream 11
mailer_get_url 11
mailer_open 11
mailer_send_message 11
mailer_set_debug 11
mailer_set_stream 11
mapfile_stream_create 20
memory_stream_create 20
message_create 13
message_create_attachment 14
message_destroy 13
message_encapsulate 14
message_get_attribute 14
message_get_body 13
message_get_envelope 14
message_get_header 13
message_get_num_parts 13
message_get_observable 14
message_get_part 14
message_get_stream 14
message_get_uid 14
message_get_uidl 14
message_is_multipart 13
message_save_attachment 14
message_set_attribute 14
message_set_body 13
message_set_envelope 14
message_set_header 13
message_set_stream 14
message_set_uidl 14
message_unencapsulate 14
mu-address-get-comments 41
mu-address-get-count 42
mu-address-get-domain 42
mu-address-get-email 41
mu-address-get-local 42
mu-address-get-personal 41
mu-closelog 44
mu-logger 44
mu-mailbox-append-message 42
mu-mailbox-close 42
mu-mailbox-expunge 42

mu-mailbox-get-message 42
mu-mailbox-get-port 42
mu-mailbox-get-url 42
mu-mailbox-messages-count 42
mu-mailbox-open 42
mu-mailbox-url 42
mu-message-copy 43
mu-message-delete 43
mu-message-get-body 44
mu-message-get-flag 43
mu-message-get-header 43
mu-message-get-header-fields 43
mu-message-get-lines 43
mu-message-get-port 44
mu-message-get-sender 43
mu-message-get-size 43
mu-message-get-user-flag 43
mu-message-send 44
mu-message-set-flag 43
mu-message-set-header 43
mu-message-set-header-fields 43
mu-message-set-user-flag 44
mu-mime-add-part 44
mu-mime-create 44
mu-mime-get-message 44
mu-mime-get-num-parts 44
mu-mime-get-part 44
mu-mime-multipart? 44
mu-openlog 44
mu_auth_data 37
mu_auth_data_alloc 38
mu_auth_data_free 38
mu_auth_fp 37
mu_auth_generic_module 39
mu_auth_init 38
mu_auth_module 38
mu_auth_nosupport 39
mu_auth_pam_module 39
MU_AUTH_REGISTER_ALL_MODULES 38
mu_auth_register_module 38
mu_auth_runlist 39
mu_auth_sql_module 40
mu_auth_system_module 39
mu_auth_virtual_module 40
mu_authenticate 39
mu_get_auth_by_name 39
mu_get_auth_by_uid 39
MU_STREAM_APPEND 19
MU_STREAM_CREAT 19
MU_STREAM_NO_CHECK 19
MU_STREAM_NO_CLOSE 19
MU_STREAM_NONBLOCK 19
MU_STREAM_RDWR 19
MU_STREAM_READ 19
MU_STREAM_WRITE 19

N

numaddr 64

P

parse822_addr_spec 36
parse822_address 35
parse822_address_list 35
parse822_domain 36
parse822_domain_literal 36
parse822_domain_ref 36
parse822_field_body 36
parse822_field_name 36
parse822_group 35
parse822_local_part 36
parse822_mail_box 35
parse822_quote_local_part 36
parse822_quote_string 36
parse822_route 36
parse822_route_addr 35
parse822_sub_domain 36
parse822_unix_mbox 36

R

redirect 67
reject 65

S

sieve_abort 52
sieve_action_log_t 48
sieve_action_lookup 53
sieve_comparator_t 49
sieve_compile 54
sieve_data_type 46
sieve_debug 52
sieve_destructor_t 49
sieve_disass 54
sieve_error 52
sieve_get_daemon_email 51
sieve_get_debug_level 50
sieve_get_mailer 50
sieve_get_message 50
sieve_get_message_num 50
sieve_get_ticket 50
sieve_handler_t 48
sieve_is_dry_run 52
sieve_load_ext 53
sieve_log_action 52
sieve_machine_add_destructor 50
sieve_machine_destroy 49
sieve_machine_init 49
sieve_machine_t 46
sieve_mailbox 54
sieve_malloc 53
sieve_message 54
sieve_mfree 54

sieve_mrealloc 53
sieve_mstrdup 53
sieve_parse_error_t 48
sieve_printf_t 48
sieve_register_action 53
sieve_register_comparator 53
sieve_register_test 53
sieve_retrieve_t 49
sieve_runtime_tag_t 47
sieve_set_daemon_email 52
sieve_set_debug 51
sieve_set_debug_level 51
sieve_set_error 51
sieve_set_logger 52
sieve_set_mailer 52
sieve_set_parse_error 51
sieve_set_ticket 52
sieve_tag_checker_t 49
sieve_tag_def_t 47
sieve_tag_lookup 53
sieve_test_lookup 53
sieve_type_str 52
sieve_value_t 46
size 62
stdio_stream_create 20
stop 65
stream_close 20
stream_create 21
stream_destroy 20
stream_flush 20
stream_get_fd 20
stream_get_flags 21
stream_get_state 21
stream_is_seekable 20
stream_open 20
stream_read 20
stream_readline 20
stream_setbufsiz 20
stream_size 20
stream_truncate 20
stream_write 20
string_to_attribute 19

T

tcp_stream_create 20
true 62

U

url_create 33
url_decode 34
url_destroy 33
url_get_host 33
url_get_passwd 33
url_get_path 34
url_get_port 34
url_get_query 34
url_get_scheme 33
url_get_user 33
url_parse 33

Program Index

C

comsatd 114

F

frm 82

G

guimb 111

I

imap4d 79

M

mail 83
mail.local 102
mailutils-config 118
messages 105

P

pop3d 81

R

readmsg 106

S

sieve 107

Concept Index

This is a general index of all issues discussed in this manual

#

#include, sieve ... 58
#searchpath, sieve ... 59

:

:address ... 75
:auth ... 76
:daemon ... 76
:logging ... 77
:mailbox ... 75
:mailer ... 75
:sieve ... 77

A

Acknowledgement ... 121
Address ... 25
Attribute ... 18
authentication ... 72
Authentication Library ... 36
Authenticator ... 24
authorization ... 72

B

Body ... 17

E

Envelope ... 15

F

Folder ... 4
Framework ... 2

H

Headers ... 15

I

IMAP4 namespace ... 79
Introduction ... 1
Iterator ... 23

L

libmu_scm ... 41
libmuauth ... 36
libmuauth modules ... 39
libmuauth, data types ... 37
libmuauth, linking with ... 40
libmuauth, obtaining authorization information ... 39
libsieve ... 45
linking with libmuauth ... 40
Locker ... 31

M

Mailbox ... 6
Mailer ... 11
Mailutils configuration file ... 74
mailutils.rc ... 74
mailutils.rc, an example ... 78
Message ... 13

N

namespace ... 79

P

Programs ... 71

R

Reporting Bugs ... 121

S

Scheme ... 41
Sieve Library ... 45
Sieve preprocessor statements, a GNU extension ... 58
Stream ... 19

U

URL ... 32
using libmuauth ... 40

Table of Contents

1 Introduction 1

1.1 References 1

2 Framework 2

2.1 Folder 4
2.2 Mailbox 6
2.3 Mailer 11
2.4 Message 13
2.5 Envelope 15
2.6 Headers 15
2.7 Body 17
2.8 Attribute 18
2.9 Stream 19
2.10 Iterator 23
2.11 Authenticator 24
2.12 Address 25
2.13 Example 28
2.14 Locker 31
2.15 URL 32
2.15.1 POP3 32
2.15.2 IMAP 32
2.15.3 File 32
2.15.4 Mailto 33
2.15.5 URL functions 33
2.16 Example 34
2.17 Parse822 — Functions for Parsing RFC 822 Headers 35

3 Authentication Library 36

3.1 Data Types 37
3.2 Initializing the libmuauth 38
3.3 Module Creation and Destruction 38
3.4 Obtaining Authorization Information 39
3.5 Existing Modules 39
3.6 Using Libmuauth in Your Programs 40

4 Mailutils to Scheme interface 41

4.1 Address Functions 41
4.2 Mailbox Functions 42
4.3 Message Functions 43
4.4 MIME Functions 44
4.5 Log Functions 44
4.6 Direct Linking 45
4.7 Dynamic Linking 45

5 Sieve Library 45

5.1 Library Description 45
5.1.1 Sieve Data Types 46
5.1.2 Manipulating the Sieve Machine 49
5.1.3 Logging and Diagnostic Functions 52
5.1.4 Symbol Space Functions 53
5.1.5 Memory Allocation 53
5.1.6 Compiling and Executing the Script 54
5.2 Input Language 54
5.2.1 Lexical Structure 54
5.2.2 Syntax 56
5.2.2.1 Commands 56
5.2.2.2 Actions Described 57
5.2.2.3 Control Flow 57
5.2.2.4 Tests and Conditions 58
5.2.3 Preprocessor 58
5.2.3.1 Sieve #include directive 58
5.2.3.2 Sieve #searchpath directive 59
5.2.4 Require Statement 59
5.2.5 Comparators 60
5.2.6 Tests 60
5.2.7 Actions 65
5.3 Extensions 67
5.4 Writing Loadable Commands 68

6 Programs 71

6.1 Authorization and authentication principles 72
6.2 Mailutils configuration file 74
6.2.1 default — Options understood by most GNU utilities. 74
6.2.2 mailbox — Specifies the mail spool location, and locking strategy. 75
6.2.3 mailer — Sets the default mailer URL. 75
6.2.4 address — Specifies the default email address and domain. 75
6.2.5 daemon — Options common for daemon programs. 76
6.2.6 auth — Authentication-specific options. 76
6.2.7 logging — Logging control options 77
6.2.8 Sieve specific options 77
6.2.9 A sample configuration file 78
6.3 IMAP4 daemon 79
6.3.1 Imap4d Namespace 79
6.3.2 Starting imap4d 79
6.4 POP3 daemon 81
6.5 frm — List headers from a mailbox 82
6.6 mail — send and receive mail 83
6.6.1 Command line options 83

6.6.2 How to specify message sets 84
6.6.3 Composing mail 86
6.6.3.1 Quitting Compose Mode 86
6.6.3.2 Getting Help on Compose Escapes: ~? . . 86
6.6.3.3 Editing the Message: ~e and ~v. 86
6.6.3.4 Modifying the Headers: ~h, ~t, ~c, ~b, ~s 86
6.6.3.5 Enclosing Another Message: ~m and ~M 87
6.6.3.6 Adding a File to the Message: ~r and ~d 87
6.6.3.7 Printing And Saving the Message 87
6.6.3.8 Signing the Message: ~a and ~A 87
6.6.3.9 Printing Another Message: ~f and ~F . . . 87
6.6.3.10 Inserting Value of a Mail Variable: ~i . . 88
6.6.3.11 Executing Other Mail Commands: ~: and ~- 88
6.6.3.12 Executing Shell Commands: ~! and ~| 88
6.6.4 Reading mail 88
6.6.4.1 Quitting the program 89
6.6.4.2 Obtaining online help 89
6.6.4.3 Moving within a mailbox 89
6.6.4.4 Changing mailbox/directory 89
6.6.4.5 Controlling header display 90
6.6.4.6 Displaying information 90
6.6.4.7 Displaying messages 91
6.6.4.8 Marking messages 92
6.6.4.9 Disposing of messages 92
6.6.4.10 Saving messages 93
6.6.4.11 Editing messages 94
6.6.4.12 Scripting 94
6.6.4.13 Aliasing 96
6.6.4.14 Replying 96
6.6.4.15 Incorporating new mail 97
6.6.4.16 Shell escapes 97
6.6.5 How to alter the behavior of mail 97
6.6.6 Personal and system-wide configuration files 101
6.7 mail.local — Deliver mail to the local mailbox. 102
6.7.1 Invoking mail.local 102
6.7.2 Using mail.local with various MTAs 103
6.7.2.1 Using mail.local with sendmail. 103
6.7.2.2 Using mail.local with exim. 104
6.7.3 Setting up mailbox quotas 104
6.7.4 Implementing user-defined Sieve mail filters. 104
6.7.5 Implementing user-defined Scheme mail filters. . . 104
6.8 messages — Count the number of messages in a mailbox. 105

6.9 readmsg — Extract messages from a folder. 106
6.10 sieve 107
6.10.1 A Sieve Interpreter 107
6.10.1.1 Invocation 107
6.10.1.2 Logging and debugging 108
6.10.1.3 Extending Sieve 109
6.10.2 A Sieve to Scheme Translator and Filter 110
6.11 guimb — A mailbox scanning and processing language. . . 111
6.11.1 Specifying Scheme Program to Execute 111
6.11.2 Specifying Mailboxes to Operate Upon 111
6.11.3 Passing Options to Scheme 112
6.11.4 Guimb Invocation Summary 112
6.12 Comsat daemon 114
6.12.1 Starting comsatd 114
6.12.2 Configuring comsatd 114
6.12.2.1 General Settings 115
6.12.2.2 Security Settings 115
6.12.2.3 Access Control Lists 115
6.12.3 A per-user configuration file. 116
6.12.3.1 Example I. 116
6.12.3.2 Example II. 117
6.13 mailutils-config — Get the information about the Mailutils build. 118
6.13.1 Getting Compiler Flags. 118
6.13.2 Getting Loader Flags 118
6.13.3 Obtaining General Build Information 118

7 Reporting Bugs 121

8 Getting news about GNU mailutils 121

9 Acknowledgement 121

Function Index 122

Program Index 126

Concept Index 127