Summary:

libpostal is a mail library.

What more can be said? libpostal is intended to handle all variety of mail access and processing tasks, with a current concentration on reading from and (for thost that support it) writing to various mail storage formats. We currently have a very workable POP3 implementation, a reasonably complete (though rough around the edges) mbox implementation, and a similarly complete-though-rough Maildir implementation.

The current ongoing thrust is to improve the robustness of the extant parts of the library, with better error handling and memory management, as well as algorithmic work on the actual functional code. Then, down the line, we have our eyes on formats like MH folders, IMAP, and possibly more esoteric formats and protocols like MAPI etc.

The current section in development is libpost_raw. This is the ’raw’ interfaces to the mbox’s, POP servers, etc. Later, we plan to write a more ’refined’ interface, with a full abstraction, where after opening a ’mailbox connection’, you don’t have to worry about what kind of connection it is, just what you want to do with it. But, that’s for the future. First things first.

This document is currently a combination of a documentation of libpostal’s interfaces (i.e., "Programming using libpostal"), and of its structure and internals (i.e., "Developing libpostal"). Eventually, these will probably be split into two seperate documents. For now, just take note that certain data structures and functions are ’private’, and intended for internal usage only. These structures/functions are *NOT* specified or prototyped in the public header file, so attempting to use them should cause compilation errors, unless you have such warnings turned off. Data types or functions in this document which are intended to be private have a notice after their short description saying so. See, for instance, **postal_flock()**.

Also, bear in mind that all data structures are intended to be opaque. No program using libpostal should ever mess around inside the structures, their contents are provided only for instructional purposes, and are subject to change. A set of macros are currently provided (though undocumented, except in the sample/test programs and the header file) for such access; these will be replaced down the road by full functions, to provide a little more security against API changes.

Note: libpostal is currently still in a fairly early development phase. We don’t recommend using it for Real Work (tm) as of yet. Current releases are all intended for developers to use as reference bases, or for the world at large to use to see what we’re doing and where we’re going. Until our first true release (i.e., Version 1.0), all API’s described here are subject to possible change. You’ve Been Warned.

Contents

maildir_free_spec() (Internal)
maildir_get_message()
maildir_get_message_all()
maildir_get_status()
maildir_msg_filename_construct() (Internal)
maildir_msg_filename_dissect() (Internal)
maildir_open()
maildir_status2info() (Internal)
maildir_write_msg()
mbox_alloc_spec() (Internal)
mbox_close_file()
mbox_derive_from() (Internal)
mbox_free_spec() (Internal)
mbox_get_message()
mbox_get_message_all()
mbox_get_message_next()
mbox_get_status()
mbox_lock_read() (Internal)
mbox_lock_write() (Internal)
mbox_open_file()
mbox_set_conlen() (Internal)
mbox_set_status() (Internal)
mbox_unlock() (Internal)
mbox_validate_from() (Internal)
mbox_write_msg()
pop_alloc_spec() (Internal)
pop_answer_check() (Internal)
pop_close()
pop_connect()
pop_dele()
pop_free_retlist()
pop_free_retstat()
pop_free_spec() (Internal)
pop_list()
pop_login()
pop_network_close() (Internal)
pop_network_connect()
pop_retr()
pop_retr_all()
pop_stat()

Data structures:

- POSTAL_CONN: A single libpost_raw mailbox connection

  Note:

  Use the **postal_free_conn()** function to free the memory associated with this datatype, and the **postal_alloc_conn()** function to allocate and initialize the structure.

  Members:

  - union spec: Special connection-type-specific info
    - **SPEC_POP** * pop: A POP connection
    - **SPEC_MBOX** * mbox: A mbox connection
    - **SPEC_MDIR** * mdir: A Maildir connection
  - int conn_type: A flag denoting the type of mailbox connected to
  - **POSTAL_MSG_GROUP** * msgs: Messages associated with this connection

- POSTAL_MSG: A single email message

  Note:

  Use the **postal_free_msg()** function to free the memory associated with this datatype, and the **postal_alloc_msg()** function to allocate and initialize the structure.

  Members:

  - char * header: The headers of the email
  - size_t hdrlen: Length of the header
  - char * body: The body of the email
  - size_t bodylen: Length of the body
  - union loc: The location of the message
    - **MAILDIR_MSG** * file: A file location (as in a Maildir/)
    - off_t offset: A file offset location (as in a mbox)
  - int src_type: A flag indicating what mailbox type the message came from
  - int stat_loc: The current status of the message in the mailbox
  - int stat_new: The current status of the message in memory (which will need to be sync'd into the mailbox)
  - int dirty: A flag for indicating 'clean/dirty' status

- POSTAL_MSG_GROUP: A group of email messages

Note:

Use the **postal_alloc_msggroup()** function to initialize this structure. Use the **postal_free_msggroup()** function to free the memory associated with this datatype.

Members:

- **POSTAL_MSG** ** msgs: A list of all the messages. Done as a ** so it can be referenced as an array.

- long num_msgs: The number of messages in the group. The last message (remember your arrays!) is ->msgs[num_msgs-1].

- long alloced: The number of messages space is allocated for in the group.

- MAILDIR_MSG: A Maildir message location

Note:

This is used as a member of the ->loc union in the **POSTAL_MSG** structure.
Use the **maildir_free_mdmsg()** function to free the memory associated with this datatype, and the **maildir_alloc_mdmsg()** function to allocate and initialize the structure.

Members:

- char * filename: The full filename of the message

- char * unique: The ’unique’ portion of the filename of the message

- off_t size: The message size, if specified in the filename. Courier (MTA, IMAP, POP, maildrop) does this, and it’s a useful thing.

- char * info: The info portion of the filename of the message

- int loc: The location of the message (LOC_TMP || LOC_NEW || LOC_CUR)

- char * obase: The Maildir base directory that we came from

- POP_RET_LIST: The results of a POP ’LIST’ query

Note:

Use the **pop_free_retlist()** function to free the memory associated with this datatype.

Members:

- unsigned int msg_num: Message number

- unsigned long msg_size: Size of the message

- **POP_RET_LIST** * next: Structure for next message

- POP_RET_STAT: The results of a POP ’STAT’ query

Note:

This is subject to inherent race conditions, since you have no guarantee that no new mail has been delivered since you queried, unless the POP server itself locks the mailbox and keeps a static view of what’s in it throughout the POP session (most do).

Use the **pop_free_retstat()** function to free the memory associated with this datatype.

Members:

- unsigned int num_msg: Number of messages available on the server
- unsigned long num_bytes: Total size of all available messages

- SPEC_MBOX: Special info for a mbox connection

Note:

This is used as a member of the ->spec union in the **POSTAL_CONN** structure.

Use the **mbox_free_spec()** function to free the memory associated with this datatype, and the **mbox_alloc_spec()** function to allocate and initialize the structure.

Members:

- char * filename: The filename of the mbox
- int locks: A bitmask of the types of file locks applied
- FILE * file: The stdio stream associated with the mbox
- int desc: The file descriptorassociated with the mbox

- SPEC_MDIR: Special info for a maildir connection

Note:

This is used as a member of the ->spec union in the **POSTAL_CONN** structure.

Use the **maildir_free_spec()** function to free the memory associated with this datatype, and the **maildir_alloc_spec()** function to allocate and initialize the structure.

Members:

- char * base: The base directory of the Maildir/

- SPEC_POP: Special info for a POP connection

  Note:

  This is used as a member of the ->spec union in the **POSTAL_CONN** structure.

  Use the **pop_free_spec()** function to free the memory associated with this datatype, and the **pop_alloc_spec()** function to allocate and initialize the structure.

  Members:

  - int sock: The socket descriptor for the connection to the server
  - char * srv_header: The contents of the server ’banner’

- POSTAL_ERRFOO_T: Error codes and details

  *** This datatype is for internal use only ***

  Note:

  This is used slightly differently in the normal and pthreads variants of the library, but in neither case should it ever be touched directly. If you’re setting values in it (which should only happen inside the library), use **postal_err_set()**. If you’re checking errors from a user program, use **postal_errno()** and **postal_errstr()**. Never manipulate, extern, declare, or otherwise use this datatype.

  Members:

  - unsigned int postal_errno: A numeric code for the error. May also hold a constant value noting that there is a character string also available with details. Only **postal_errstr()** knows for sure.
  - char * postal_errstr: A string containing details

Functions:

- postal_add_header(): Add a header into a message

Summary:

Takes a message and a header line, and adds that header line into the header section of the message.

Note that this does no duplicate checking or any other similar things. It just adds the header onto the end of the header block.

Arguments:

- **POSTAL_MSG** * msg: A single email message
- const char * toadd: A header line (with no trailing newline) to add

Return value:

int

Returns 0 on success. Returns -1 on error.

Error codes:

- [POSTAL_E_INVAL]
  Some part of the arguments was invalid
- [POSTAL_E_NOMEM]
  Memory allocation failure

- postal_alloc_msggroup(): Allocate a **POSTAL_MSG_GROUP**

Summary:

Allocate and initialize a **POSTAL_MSG_GROUP** for use by the program.

Remember to use the **postal_free_msggroup()** function to free the structure when you're done with it.

Arguments:

None.

Return value:

POSTAL_MSG_GROUP *

Returns a pointer to an initialized **POSTAL_MSG_GROUP** on success. Returns NULL on failure.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure

- postal_change_header(): Change a header in a message

Summary:

Takes a message, a header name, and a header content. Finds the current instance (or the first current, if more than one) of that header, and adjusts its content to the supplied value.

NOTE: The hdr argument should include the terminating colon, but no space after it. Thus, "From:", but not "From" or "From: ".

NOTE: This function only works with headers that already exist. If the header doesn’t exist, use **postal_add_header()**.

Arguments:

- **POSTAL_MSG** * msg: A single email message
- const char * hdr: The header to replace
- const char * content: The content to place in the header

Return value:

int

Returns 0 on success. Returns -1 on error.

Error codes:

- [POSTAL_E_INVAL]
  Some part of the arguments was invalid
- [POSTAL_E_NOMEM]
  Memory allocation failure
- [POSTAL_E_NOHDR]
  Header not found
- [POSTAL_E_DFORM]
  Data format error: malformed headers

- postal_del_header(): Delete a header from a message

Summary:

Takes a message and a header line, and deletes that header line from the header section of the message.

Note that this does no duplicate checking or any other similar things. It just deletes the first instance of the specified header.

Arguments:

- **POSTAL_MSG** * msg: A single email message
- const char * hdr: A header name (with no trailing newline) to delete

Return value:

int

Returns 0 on success. Returns -1 on error.

Error codes:

- [POSTAL_E_INVAL]
  Some part of the arguments was invalid
- [POSTAL_E_NOMEM]
  Memory allocation failure
- [POSTAL_E_NOHDR]
  Header not found
- [POSTAL_E_DFORM]
  Data format error: malformed headers

- postal_errno(): Get numeric error code

Summary:

Returns a numeric code representing the last error encountered inside the library. This is the moral equivalent of C's errno facility. If no error has occured, its value is indeterminate.

Arguments:

None.

Return value:

unsigned int

Returns a numeric error code. See individual functions for descriptions of what error codes each may set.

- postal_errstr(): Get error details as an opaque string

Summary:

Sometimes an error may have additional information available in a freeform string. If so, this function will return it. If no error has occured, or no string value has been set for an error, the value is indeterminate.

In practice, this is rarely set, and even more rarely of any real use to anybody. This is provided because occasionally there might be a gem, and in the future we might want to expand use of this. Right now, unless you're really anal, it's probably never worth calling this function.

Arguments:

None.

Return value:

const char *

If a string error has been set, return a pointer to it. If not, return NULL.

NOTE: Do NOT attempt to **free()** or otherwise manipulate the string returned. It's the private property of **postal_errstr()**. Read from it, or use it as the source of a **strdup()** or **strcpy()** or in a **printf()** format string, fine. Try to write into it or **free()** it or **realloc()** it or some such, and goblins will hunt you down and kill you in your sleep. Or your program will SIGBUS and laugh at you. Whichever.

- postal_free_msggroup(): Free a **POSTAL_MSG_GROUP** structure

Summary:

Takes a **POSTAL_MSG_GROUP** structure and **free()**'s all its component parts.

Arguments:

- **POSTAL_MSG_GROUP** * tofree: A single **POSTAL_MSG_GROUP** structure.

Return value:

void

- postal_get_header(): Returns a header's contents

Summary:

Searches through the given headers for the requested header and returns its contents.

NOTE: The hdr argument should include the terminating colon, but no space after it. Thus, "From:", but not "From" or "From: ".

Arguments:

- const char * hdrs: A set of mail headers
- const char * hdr: A header name

Return value:

char *

If the header is found, a string is returned containing the contents of the header.

Example:

(headers)

```
...
From: Me <me@some.where>
...
(end)
```

postal_get_header(msg->header, "From:") will return a string containing "Me <me@some.where>".

If the requested header is not found, NULL will be returned and the error code will be set to POSTAL_E_NOERR. If some other error occurs, NULL will be returned and the error code set to something else.

Error codes:

- [POSTAL_E_INVAL]
  Invalid argument
- [POSTAL_E_NOMEM]
  Memory allocation failure
- [POSTAL_E_DFORM]
  Data format error: malformed headers

- postal_pthread_init(): Initialize pthreads stuff for a process

Summary:

Setup necessary bits and pieces for a thread in a pthreads-enabled process using libpostal. This is only available in the pthreads variant of the library. This function (and the pthreads variant itself) should only be used when you’re using libpostal functions in multiple threads simultaneously, in which case it will keep them from stomping on each other’s error messages.

This function should be called once in the process, before any threads using libpostal are spawned off. See also **postal_pthread_thread_init()** and **postal_pthread_thread_fini()**.

Arguments:

None.

Return value:

int

Returns 0 on success. Returns -1 on error (errors come from **pthread_key_create()**)

- postal_pthread_thread_fini(): Destroy pthreads stuff for a thread

Summary:

Destroy necessary bits and pieces for a thread in a pthreads-enabled process using libpostal. This is only available in the pthreads variant of the library. This function (and the pthreads variant itself) should only be used when you’re using libpostal functions in multiple threads simultaneously, in which case it will keep them from stomping on each other’s error messages.

This function should be called once in each thread using libpostal functions, after finished calling any of them. If you fail to call this function before destroying the thread, you’ll leak memory. See also **postal_pthread_thread_init()**.

Arguments:

None.

Return value:

void

- postal_pthread_thread_init(): Initialize pthreads stuff for a thread

Summary:

Setup necessary bits and pieces for a pthreads-enabled process using libpostal. This is only available in the pthreads variant of the library. This function (and the pthreads variant itself) should only be used when you’re using libpostal functions in multiple threads simultaneously, in which case it will keep them from stomping on each other’s error messages.

This function should be called once in each thread using libpostal functions, because any of them are called. See also **postal_pthread_thread_fini()**.

Arguments:

None.

Return value:

int

Returns 0 on success. Returns -1 on error from **malloc()**. Returns -2 on error from **pthread_setspecific()**.

- postal_set_header(): Set a header in a message

Summary:

Takes a message, a header name, and a header content. Searches through the headers of the message; if header already exists, call **postal_change_header()** to change it to the given value. If header doesn’t exist, call **postal_add_header()** to add the value in.

NOTE: The hdr argument should include the terminating colon, but no space after it. Thus, "From:", but not "From" or "From: ".

Arguments:

- **POSTAL_MSG** * msg: A single email message
- const char * hdr: The header to set
- const char * content: The content to place in the header

Return value:

int

Returns 0 on success. Returns non-zero on error (no current cases).

Error codes:

- [POSTAL_E_INVAL]
  Invalid argument
- [POSTAL_E_NOMEM]
  Memory allocation failure

Note:

postal_set_header() can also fail and return any error codes specified for **postal_get_header()** or **postal_add_header()**

- postal_strerror(): Get a friendly description of the error

Summary:

An error number from **postal_errno()** is precise, but not too descriptive. This function lets you get a more useful string out of it for presenting to a user. It's the moral equivalent of your system's **strerror()** function.

Arguments:

- int p_errno: The error number to look up. Generally, the return code of **postal_errno()**.

Return value:

const char *

A string describing the error.

NOTE: Do NOT attempt to **free()** or otherwise manipulate the string returned.

- maildir_close(): Close a Maildir connection

Summary:

This function is passed an open maildir connection structure. It **free()**'s the memory associated with the structure. The obvious counterpart to **maildir_open()**.

Arguments:

- **POSTAL_CONN** * p_conn: A Maildir connection to close down

Return value:

void

- maildir_create(): Create a Maildir in the filesystem

Summary:

This function takes a filesystem location as an argument, and constructs (if possible) a full valid Maildir there.

Arguments:

- char * where: Where to make the Maildir

Return value:

int

0 if successful. -1 on error.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure
- [POSTAL_E_INVAL]
  Invalid argument: Not a directory
- [POSTAL_E_FSOP]
  Filesystem error: Can’t create directory

- maildir_get_message(): Gets a single message from a Maildir

Summary:

This function retrieves a given message from a Maildir.

The message to get is determined by a combination of the Maildir currently open in the given **POSTAL_CONN**, and the individual file indicated in the **POSTAL_MSG**.

This function is called internally by **maildir_get_message_all()** when called in GET_HEADER or GET_BODY modes. You can also call this function directly using a **POSTAL_MSG** populated either manually (tricky!) or by a previous call to **maildir_get_message_all()** in GET_OFFSET or similar mode.

Arguments:

- **POSTAL_CONN** * p_conn: An open Maildir mailbox connection
- **POSTAL_MSG** * msg: The message to retrieve
- int get_type: What to get (GET_HEADER or GET_BODY)

Return value:

int

0 on success. -1 on error.

Error codes:

- [POSTAL_E_INVAL]
  Invalid argument: GET_OFFSET requested, message not from Maildir
- [POSTAL_E_INVAL]
  Invalid argument: GET_OFFSET requested, no filename set
- [POSTAL_E_NOMEM]
  Memory allocation failure
- [POSTAL_E_DFORM]
  Data format error: Bad headers
- [POSTAL_E_FSOP]
  Filesystem error: Can’t stat message file
- [POSTAL_E_FSOP]
  Filesystem error: Can’t open message file
- [POSTAL_E_FSOP]
  Filesystem error: Failed reading from message file

- maildir_get_message_all(): Gets all messages from a Maildir

Summary:

This function retrieves all messages from a Maildir.

This functions works by calling the **postal_list_dir()** function, then iteratively calling **maildir_get_message()** for each message in the Maildir, cur/ and then new/.

However, if called in GET_OFFSET mode, it will only create and populate the **POSTAL_MSG_GROUP** within the open **POSTAL_CONN** for the Maildir in question, and not call **maildir_get_message()** for it. When called in this mode, the **POSTAL_MSG** items created in the **POSTAL_MSG_GROUP** can be used for later direct calls to **maildir_get_message()**.

Arguments:

- **POSTAL_CONN** * p_conn: An open Maildir mailbox connection
- int get_type: What to get (GET_OFFSET || GET_HEADER || GET_BODY)

Return value:

long

Number of messages retrieved on success. -1 on error.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure

Note:

maildir_get_message_all() may also fail and set an error for any of the codes listed for **postal_list_dir()**, **maildir_get_message()**, **maildir_msg_filename_dissect()**, or **postal_alloc_msg()**.

- maildir_get_status(): Get message status

Summary:

This function allows getting the ’status’ of a message from a Maildir. It currently supports the following statae:

STATUS_NULL: Error (no message, uninitialized status, etc
STATUS_NEW: A new message
STATUS_OLD: A not-new message
STATUS_READ: A message that has been ’read’
STATUS_REPLIED: A message that has been replied to
The precise meanings of these flags will vary depending on the mail client’s precise semantics.

The value returned is a bitmask, based on which flags are set. So, to test for ’read’ status, you’d use a construct like:

```
status = maildir_get_status(msg);
if( (status & STATUS_READ) )
/* Message is read */
```

The calling and returning semantics of this function are intentionally nigh-on identical to those of the **mbox_get_status()** function.

Arguments:

- **POSTAL_MSG** * msg: A single mail message (with populated header)

Return value:

int

Returns a bitmask of the statae that exist on the message. Returns -1 on error.

- maildir_open(): Open a Maildir connection

Summary:

This function opens (and verified) a Maildir as a POSTAL_CONN connection.

If the ’create’ argument is 1, **maildir_open()** will call **maildir_create()** internally to create the Maildir. Otherwise, it will return an error if the Maildir isn’t pre-created.

Arguments:

- char * where: Where to make the Maildir
- int create: Create if nonexistent

Return value:

POSTAL_CONN *

Populated **POSTAL_CONN** structure on success. NULL on error

Error codes:

- [POSTAL_E_INVAL]
  Invalid argument: Not a directory
- [POSTAL_E_NOMEM]
  Memory allocation failure
- [POSTAL_E_NEXIST]
  Directory doesn’t exist

Note:

> **maildir_open()** may also fail and set an error for any of the codes listed for **maildir_create()**.

- maildir_write_msg(): Write a message into a Maildir

Summary:

Writes a given message into a given Maildir connection.

This function internally double-checks things like \r\n translation. This doesn’t mean you should rely on it doing those things for you; just that it double-checks.

It also handles changing around filenames for status changes, size changes, and all that jazz. You SHOULD rely on it to do that; don’t mess with the internals yourself!

Arguments:

- **POSTAL_CONN** * p_conn: An open Maildir connection
- **POSTAL_MSG** * msg: The message to write

Return value:

int

0 on success. -1 on error.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure
- [POSTAL_E_FSOP]
  Filesystem operation failure: Can’t stat()
- [POSTAL_E_FSOP]
  Filesystem operation failure: rename() failed
- [POSTAL_E_FSOP]
  Filesystem operation failure: fopen() failed
- [POSTAL_E_EXIST]
  Destination file already exists

Note:

> **maildir_write_msg()** can also fail and return any error codes specified for **postal_whack_cr()**, **postal_del_header()**, **postal_set_header()**, **postal_get_header()**, **maildir_msg_filename_construct()**, **maildir_msg_filename_dissect()**, **maildir_status2info()**.

- mbox_close_file(): Close a mbox connection

Summary:

This function is passed an open mbox connection structure. It flushes and closes all associated files, and **free()**'s the memory associated with the structure.

Arguments:

- **POSTAL_CONN** * p_conn: A mbox connection to close down

Return value:

void

- mbox_get_message(): Get a message from a mbox

Summary:

This function is used after a mbox has been scanned, and you've gotten a **POSTAL_MSG** partially populated, with at least an offset, and possibly the headers, of a message. It then retrieves the headers (and possibly the body, depending on what you request) for the associated message.

It's also used internally by the **mbox_get_message_next()** function to get messages by faking up a previously-grabbed message structure. Don't try this at home, kids.

Its primary use is to MUA's which will, for instance, scan a mbox and get the offsets and headers for each message (to build a list and have a message 'index'), but will NOT load the bodies of the messages for memory usage purposes. **mbox_get_message()** can be used to then quickly grab the body of a given message.

What the function grabs depends on the contents of the get_type argument. It should be one of the following:

GET_HEADER: Get the header of the message
GET_BODY: Get the body (and header) of the message
Each type includes the type before, so GET_HEADER will get offsets as well, and GET_BODY will also gets offsets and headers. If the get_type specifies to get a portion of the message that already exists in the passed message structure, the old contents will be overwritten.

Note: This is a very simplistic function as things currently stand. It has no real provisions for handling the mbox having changed out from under it since the **POSTAL_MSG** structure was originally populated. We’ll be making a slightly more all-encompassing method of handling this somewhere down the road. mbox’s suck.

Arguments:

- **POSTAL_CONN** * p_conn: The mbox connection to look up the message in
- **POSTAL_MSG** * msg: The message to look up
- int get_type: The type of lookup to make.

Return value:

int

0 if successful. -1 on error.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure
- [POSTAL_E_DFORM]
  Data format error: read() hit EOF unexpectedly
- [POSTAL_E_FSOP]
  Filesystem operation error: read() failed
- [POSTAL_E_NOMSG]
  No message collected (end of file before we got anything). This may well not be an error; for instance, if we called **mbox_get_message_next()** after we’d already gotten the last message from the mbox. This is distinct from the POSTAL_E_DFORM return in that here, it’s possible to correctly hit EOF, while with POSTAL_E_DFORM, it’s definately a Bad Thing to EOF.

Note:

mbox_get_message() can also fail and return any error codes specified for **postal_get_header()** or **mbox_get_status()**.

- mbox_get_message_all(): Gets all messages from a mbox

Summary:

mbox_get_message_all() is passed an open mbox connection. It retrieves all the messages in that mbox, and puts them in the **POSTAL_CONN** that it’s passed. Depending on the value of the get_type argument, the **POSTAL_MSG_GROUP** may contain just the offsets of each message, offsets and headers, or the offsets, header, and full body.

Arguments:

- **POSTAL_CONN** * p_conn: An active mbox connection to load messages from
- int get_type: The type of loading to do.

Return value:

long

Returns the number of messages read, or -1 on error.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure
- [POSTAL_E_DFORM]
  Data format error: read() hit EOF unexpectedly
- [POSTAL_E_FSOP]
  Filesystem operation error: read() failed

Note:

mbox_get_message_all() can also fail and return any error codes specified for **postal_get_header()**, **postal_alloc_msg()**, or **mbox_get_status()**.

- mbox_get_message_next(): Get the next message in a mbox

Summary:

This function currently assumes that the current ’position’ in the mbox is the beginning of a new message. We should probably fix that.

This function retrieves and returns the ’next’ message in the mbox. It uses the **mbox_get_message()** function on the backend to do the actual work.

What the function grabs depends on the contents of the get_type argument. It should be one of the following:

GET_OFFSET: Get the offsets of each message
GET_HEADER: Get the header of the message
GET_BODY: Get the body of the message
Each type includes the type before, so GET_HEADER will get offsets as well, and GET_BODY will also gets offsets and headers.

Arguments:

- **POSTAL_CONN** * p_conn: Active mbox connection
- int get_type: What to get for each message

Return value:

POSTAL_MSG *

Returns a populated **POSTAL_MSG** structure on success. Returns NULL on error.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure

Note:

mbox_get_message_next() can also fail and return any error codes specified for **mbox_get_message()**.

- mbox_get_status(): Get message status

Summary:

This function allows getting the 'status' of a message in a mbox. It currently supports the following statae:

STATUS_OLD: A not-new message
STATUS_READ: A message that has been 'read'
STATUS_REPLIED: A message that has been replied to
The precise meanings of these flags will vary depending on the mail client's precise semantics.

The value returned is a bitmask, based on which flags are set. So, to test for 'read' status, you'd use a construct like:

```
status = mbox_get_status(msg);
if( (status & STATUS_READ) )
/* Message is read */
```

Arguments:

- **POSTAL_MSG** * msg: A single mail message (with populated header)

Return value:

int

Returns a bitmask of the statae that exist on the message. Returns -1 on error.

Error codes:

- [POSTAL_E_INVAL]
  Bad arguments: No message header

Note:

mbox_get_status() can also fail and return any error codes specified for **postal_get_header()**.

- mbox_open_file(): Open a mbox connection

Summary:

Well, just what it says, for cryin' out loud. Open a file as a mbox connection, and prepare it for whatever we're going to do to it. The position is set to the beginning of the file.

The file can be opened with either OPEN_READONLY or OPEN_READWRITE as the method flag. If the file is not opened

READWRITE, it will be created if it doesn't already exist. In the future, checks may be added to double-check that a pre-existing file is really in mbox format.

Currently, there are 3 locktypes implemented: LOCK_FLOCK, LOCK_DOTLOCK, and LOCK_FCNTL. They do pretty much what you'd expect them to do. Further, there is an additional locktype, LOCK_DEFAULT, which defines a sort of default. If you don't have a specific reason to need whatever it is that you need, use LOCK_DEFAULT (which currently applies LOCK_FCNTL and LOCK_DOTLOCK).

Arguments:

- const char * filename: The filename of the mbox to open
- int locktype: A bitmap of the types of lock to apply
- int method: The opening method (see OPEN_* above)

Return value:

POSTAL_CONN *

Returns a populated **POSTAL_CONN** structure for the mbox connection on success. Returns NULL on failure.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure
- [POSTAL_E_FSOP]
  Filesystem operation error: open() failed
- [POSTAL_E_FSOP]
  Filesystem operation error: fdopen() failed

Note:

mbox_open_file() can also fail and return any error codes specified for **mbox_lock_read()** or **mbox_lock_write()**.

- mbox_write_msg(): Write a message into a mbox

Summary:

Writes a given message into a given mbox connection. Currently, there are two choices of location:

POS_CUR: The current position
POS_END: End of file

This function internally double-checks things like \r\n translation, and the derivation of the From_ line. This doesn't mean you should rely on the function doing those things for you; just that it double-checks.

Arguments:

- **POSTAL_CONN** * p_conn: An open mbox connection
- **POSTAL_MSG** * msg: The message to write
- int whence: Where to write the message (see POS_* above)

Return value:

int

0 if successful. -1 on error.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure

Note:

mbox_write_msg() can also fail and return any error codes specified for **mbox_derive_from()**, **mbox_set_conlen()**, **mbox_set_status()**, **postal_whack_cr()**, or **postal_alloc_msg()**.

- pop_close(): POP3 connection closer

Summary:

This function handles all the goobleygook involved in closing an active POP3 mailbox connection. It closes the appropriate network connection nicely (i.e., a real logout, not just a **close()**), and **free()**'s all memory resources associated with it.

Arguments:

- **POSTAL_CONN** * to_close: Active POP3 mailbox connection to close

Return value:

void

- pop_connect(): POP3 connection wrapper

Summary:

This is the normal entry point to connect to a POP server. It accepts a hostname to connect to, and a username/password pair for authentication.

It internally does a hostname lookup, and calls the **pop_network_connect()** and **pop_login()** functions to do the grunt work of connecting and authenticating.

Arguments:

- const char * tohost: Hostname to connect to
- const char * user: Username to connect as
- const char * pass: Password to authenticate with

Return value:

POSTAL_CONN *

Populated **POSTAL_CONN** structure if successful. NULL on failure.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure
- [POSTAL_E_NEXIST]
  Failed to lookup hostname

Note:

> **pop_connect()** may also fail and set errors for any of the reasons listed in **pop_network_connect()** or **pop_login()**.

- pop_dele(): Delete a message

Summary:

Uses the POP3 ’DELE’ command to delete a message from the POP server.

Arguments:

- **POSTAL_CONN** * server: An active POP server connection
- int msgnum: Message number to delete

Return value:

int

Returns 0 on success. Returns -1 on error.

Error codes:

Note:

> **pop_dele()** may fail and set errors for any of the reasons listed in **pop_answer_check()**.

- pop_free_retlist(): Free a **POP_RET_LIST** structure

Summary:

POP_RET_LIST is the data type returned by the **pop_list()** function, containing the data from a POP ’LIST’ command. This function **free()**’s all the memory associated with that returned info.

Arguments:

- **POP_RET_LIST** * tofree: Pointer to the head of the list to free

Return value:

void

- pop_free_retstat(): Free a **POP_RET_STAT** structure

Summary:

Free the resources associated with the given **POP_RET_STAT** data structure.

Arguments:

- **POP_RET_STAT** * tofree: Pointer to the **POP_RET_STAT** to free

Return value:

void

- pop_list(): Run a POP 'LIST' command

Summary:

Run a POP 'LIST' query, and sort the results into a **POP_RET_LIST** structure.

Note that this returned structure should be freed with the **pop_free_retlist()** function when its usefullness has come to an end.

Arguments:

- **POSTAL_CONN** * server: An active POP server connection

Return value:

POP_RET_LIST *

Returns a **POP_RET_LIST** structure containing the results of the POP 'LIST' command. Returns NULL on error

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure
- [POSTAL_E_SOCKET]
  Socket error: Bad read()
- [POSTAL_E_SRVERR]
  Server error: Something's not OK

- pop_login(): Login to a POP server

Summary:

This function handles the ’login’ procedure to an already-connected POP server. It does *NOT* do lookups, or even **connect()** to the server.

This is INTENDED only to be used internally by the **pop_connect()** function, and not to be called directly, along with the **pop_network_connect()** function. However, this have been set as exported functions so that, in case a user of the library REALLY needs the extra flexibility, the functions are available seperately.

Arguments:

- **POSTAL_CONN** * server: An active POP server connection
- const char * username: Username to authenticate with
- const char * password: Password to authenticate with

Return value:

int

Returns 0 on success. Returns -1 on error.

Error codes:

Note:

pop_login() may fail and set errors for any of the reasons listed in **pop_answer_check()**.

- pop_network_connect(): Connect to a POP server

Summary:

This function connects to a specified POP server, but does NOT do any sort of authentication.

Like the **pop_login()** function, this is INTENDED only to be used internally by the **pop_connect()** function, and not to be called directly. However, this have been set as exported functions so that, in case a user of the library REALLY needs the extra flexibility, the functions are available seperately.

Arguments:

- struct sockaddr_in server: A sockaddr_in structure specifying the server to connect to

Return value:

POSTAL_CONN *

Returns a created **POSTAL_CONN** structure pointing at the created connection if successful. Returns NULL on error.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure
- [POSTAL_E_SOCKET]
  Socket error: socket() failed
- [POSTAL_E_SOCKET]
  Socket error: connect() failed

Note:

> **pop_network_connect()** may also fail and set errors for any of the reasons listed in **postal_get_line()**, **pop_answer_check()**.

- pop_retr(): Retrieve and parse out a message

Summary:

Retrieve a message from a POP server and parse it out into a **POSTAL_MSG** structure.

Arguments:

- **POSTAL_CONN** * server: An active POP connection
- int msgnum: Message number to retrieve

Return value:

POSTAL_MSG *

Returns a parsed-out **POSTAL_MSG** containing the requested message from the POP server. Returns NULL on error.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure
- [POSTAL_E_DFORM]
  Data format error: Bad message recieved
- [POSTAL_E_SOCKET]
  Socket error: read() failed
- [POSTAL_E_SRVERR]
  Server error: Something’s not OK

Note:

> **pop_retr()** may also fail and set errors for any of the reasons listed in **postal_alloc_msg()**.

- pop_retr_all(): Retrive all messages

Summary:

Retrieve all messages (and optionally delete them) from a POP server. Set the ’dele’ argument to 1 to delete messages, 0 to not. Build a **POSTAL_MSG_GROUP** for the results.

Arguments:

- **POSTAL_CONN** * server: An active POP connection
- int dele: Flag to delete messages

Return value:

POSTAL_MSG_GROUP *

Returns a parsed-out **POSTAL_MSG_GROUP** containing the requested message from the POP server. Returns NULL on error.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure

Note:

pop_retr_all() may also fail and set errors for any of the reasons listed in **pop_dele()**, **pop_stat()**, or **pop_retr()**.

- pop_stat(): Run a POP ’STAT’ request

Summary:

This function runs the POP3 ’STAT’ request, and returns the results, which contains the number of messages waiting on the server, and the total size of the messages.

The returned data is in a **POP_RET_STAT** structure. You should use **pop_free_retstat()** to free up the memory used by the structure when you’re finished with it.

Arguments:

- **POSTAL_CONN** * server: An active POP connection

Return value:

POP_RET_STAT *

Returns a populated **POP_RET_STAT** with the results of the ’STAT’ query. Returns NULL on error.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure
- [POSTAL_E_SOCKET]
  Socket error: read() failed
- [POSTAL_E_SRVERR]
  Server error: Unknown

- postal_alloc_conn(): Allocate a **POSTAL_CONN**

*** This function is for internal use only ***

Summary:

Allocate and initialize a **POSTAL_CONN** for use by the program.

Remember to use the **postal_free_conn()** function to free the structure when you’re done with it.

Arguments:

None.

Return value:

POSTAL_CONN *

Returns a pointer to an initialized **POSTAL_CONN** on success. Returns NULL on failure.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure

- postal_alloc_msg(): Allocate a **POSTAL_MSG**

*** This function is for internal use only ***

Summary:

Allocate and initialize a **POSTAL_MSG** for use by the program.

Remember to use the **postal_free_msg()** function to free the structure when you’re done with it.

Arguments:

None.

Return value:

POSTAL_MSG *

Returns a pointer to an initialized **POSTAL_MSG** on success. Returns NULL on failure.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure

- postal_casestr(): Case-insensitive **strstr()**

*** This function is for internal use only ***

Summary:

This function performs the equivalent of **strstr()** in a case-insensitive manner.

FreeBSD (and probably several other systems) have a **strcasestr()** function in libc to do this, but not all systems have it. The implementation provided here is derived from the FreeBSD version.

Arguments:

- char * big: String to search IN
- char * little: String to search FOR

Return value:

char *

Returns big if little is an empty string. Returns NULL if little occurs nowhere in big. Otherwise, returns a pointer to the first character of the first occurance of little within big.

- postal_dotlock(): dotlocks a file

*** This function is for internal use only ***

Summary:

This is a general-purpose function to apply a dotlock-style lock to a file.

It uses non-blocking locks, attempts a certain number (default 5) of times, with a certain pause between attempts (default 1 second). These numbers are hard-coded in global variables; we really need to come up with a better way of doing it if we're going to ever be thread-safe.

The lock type should be either LOCK_SH for 'shared', or LOCK_EX for 'exclusive'.

Arguments:

- char * tolock: Full path of the file to lock
- int type: Type of lock to apply (see above)

Return value:

int

Returns 0 on success. Returns -1 on error.

If an error is returned, no locks are applied to the file.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure
- [POSTAL_E_FSOP]
  Filesystem error: opening tempfile
- [POSTAL_E_LKFAIL]
  Failed to aquire lock

- postal_err_set(): Sets an error to be returned

*** This function is for internal use only ***

Summary:

This function allows you to set a numeric error code which will be readable by a higher level function, either part of libpostal, or part of a user program. This information will be accessed via **postal_errno()** and **postal_errstr()**.

It's the moral equivalent of C's errno facility.

Note that in the pthreads variant of the library (append '_pthread' to the lib name), this uses thread-specific data fields, so you can use libpostal functions in multiple threads at once and they won't stomp on each other's error codes. See **postal_pthread_init()**, **postal_pthread_thread_init()**, and **postal_pthread_thread_fini()** for details on the pthreads interface.

Arguments:

- int p_errno: Numeric error code to set
- const char * p_errstr: A string containing details (or NULL)

Return value:

void

- postal_fcntl_lock(): **fcntl()** locks a file

*** This function is for internal use only ***

Summary:

This is a general-purpose function to apply a **fcntl()**-style lock to a file.

It uses non-blocking locks, attempts a certain number (default 5) of times, with a certain pause between attempts (default 1 second). These numbers are hard-coded in global variables; we really need to come up with a better way of doing it if we're going to ever be thread-safe.

The lock type should be either LOCK_SH for 'shared', or LOCK_EX for 'exclusive'.

Arguments:

- int tolock: Open file descriptor of the file to lock
- int type: Type of lock to apply (see above)

Return value:

int

Returns 0 on success. Returns -1 on error.

If an error is returned, no locks are applied to the file.

Error codes:

- [POSTAL_E_LKFAIL]
  Failed to aquire lock

- postal_flock(): **flock()** locks a file

*** This function is for internal use only ***

Summary:

This is a general-purpose function to apply a **flock()**-style lock to a file.

It uses non-blocking locks, attempts a certain number (default 5) of times, with a certain pause between attempts (default 1 second). These numbers are hard-coded in global variables; we really need to come up with a better way of doing it if we're going to ever be thread-safe.

The lock type should be one of the types defined in the manpage for **flock()**. To wit, either LOCK_SH for 'shared', or LOCK_EX for 'exclusive'.

Arguments:

- int tolock: Open file descriptor of the file to lock
- int type: Type of lock to apply (see above)

Return value:

int

Returns 0 on success. Returns -1 on error.

If an error is returned, no locks are applied to the file.

Error codes:

- [POSTAL_E_LKFAIL]
  Failed to aquire lock

- postal_flush_line(): Flush a socket buffer

*** This function is for internal use only ***

Summary:

This function is given a socket descriptor, and flushes the input until a newline (\n) is reached.

Arguments:

- int sock: Socket descriptor to flush

Return value:

int

Returns 0 on success. Returns -1 on failure.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure
- [POSTAL_E_SOCKET]
  Socket error in recv() or read()

- postal_free_charpp(): Free a char **

*** This function is for internal use only ***

Summary:

Takes a char ** and **free()**'s all its component parts.

The char ** is required to be constructed so that it has a NULL entry as its 'last' item, with no NULL's prior to that, or it'll leak some memory.

Arguments:

- char ** tofree: The char ** to free.

Return value:

void

- postal_free_conn(): Free a **POSTAL_CONN** structure

*** This function is for internal use only ***

Summary:

Takes a **POSTAL_CONN** structure and **free()**'s all its component parts.

Arguments:

- **POSTAL_CONN** * tofree: A single **POSTAL_CONN** structure.

Return value:

void

- postal_free_msg(): Free a **POSTAL_MSG** structure

*** This function is for internal use only ***

Summary:

Takes a **POSTAL_MSG** structure and **free()**'s all its component parts.

Arguments:

- **POSTAL_MSG** * tofree: A single **POSTAL_MSG** structure.

Return value:

void

- postal_get_line(): Get a line from a socket

*** This function is for internal use only ***

Summary:

This function will read from a socket until it gets a newline, or until the specified character limit is reached, whichever occurs first.

Be careful that the buffer you give it will hold the number of characters you specify, since otherwise you can overflow your buffer.

Arguments:

- int sock: A socket descriptor
- char * retbuf: Buffer to read the results into
- int max: Maximum number of characters to read (+1: sizeof(retbuf))

Return value:

int

Returns 0 if specified length exceeded, or amount read if newline found.
Returns -1 on error.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure
- [POSTAL_E_SOCKET]
  Socket error in recv() or read()

- postal_list_dir(): Return a list of the regular files in a given directory.

  *** This function is for internal use only ***

  Summary:

  Flips through the given directory, and assembles a list of the files in it.

  The returned list contains only regular files (no directories, sockets, devices, symlinks, etc). It also does not recurse through any subdirectories.

  The returned char ** should be **free()**'d using the **postal_free_charpp()** function.

  Arguments:

  - const char * dir: The directory to read

  Return value:

  char **

  A list of the regular files found in the directory. If no files are found, return NULL and set **postal_errno()** to POSTAL_E_NOERR. If error occurs, return NULL with an error set as below.

  Error codes:

  - [POSTAL_E_INVAL]
    Invalid arguments
  - [POSTAL_E_NOMEM]
    Memory allocation failure
  - [POSTAL_E_FSOP]
    Filesystem operation failed

- postal_whack_cr(): Translate \r\n's to \n's

  *** This function is for internal use only ***

  Summary:

  This is used to translate network-style (or DOS-style) \r\n line-terminators to Unix-style \n's.

  A new buffer is allocated for the returned string. Note that this function does *NOT*, however, **free()** the string it's fed to process.

  Arguments:

  - char * in_str: String to translate

  Return value:

  char *

  Returns a string with all \r\n's in the input string translated to \n's. Returns NULL on failure.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure

- maildir_alloc_mdmsg(): Allocate a **MAILDIR_MSG**

*** This function is for internal use only ***

Summary:

Allocate and initialize a **MAILDIR_MSG** for use by the program.

Remember to use the **maildir_free_mdmsg()** function to free the structure when you’re done with it.

Arguments:

None.

Return value:

MAILDIR_MSG *

Returns a pointer to an initialized **MAILDIR_MSG** on success. Returns NULL on failure.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure

- maildir_alloc_spec(): Allocate a **SPEC_MDIR**

*** This function is for internal use only ***

Summary:

Allocate and initialize a **SPEC_MDIR** for use by the program.

Remember to use the **maildir_free_spec()** function to free the structure when you’re done with it.

Arguments:

None.

Return value:

SPEC_MDIR *

Returns a pointer to an initialized **SPEC_MDIR** on success. Returns NULL on failure.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure

- maildir_free_mdmsg(): Free a **MAILDIR_MSG** structure

  *** This function is for internal use only ***

  Summary:

  Takes a **MAILDIR_MSG** structure and **free()**'s all its component parts.

  Arguments:

  - **MAILDIR_MSG** * tofree: A single **MAILDIR_MSG** structure.

  Return value:

  void

- maildir_free_spec(): Free a **SPEC_MDIR** structure

  *** This function is for internal use only ***

  Summary:

  Takes a **SPEC_MDIR** structure and **free()**'s all its component parts.

  Arguments:

  - **SPEC_MDIR** * tofree: A single **SPEC_MDIR** structure.

  Return value:

  void

- maildir_msg_filename_construct(): Construct a Maildir filename

  *** This function is for internal use only ***

  Summary:

  Take in a **MAILDIR_MSG** structure containing a message in a Maildir. Build up the filename element from the other elements in the structure (unique, info, etc).

  Arguments:

  - **MAILDIR_MSG** * minfo: A structure representing a Maildir message
  - int newloc: Where the message is to end up

Return value:

int

0 for no error. -1 for error.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure

- maildir_msg_filename_dissect(): Dissect a Maildir filename

*** This function is for internal use only ***

Summary:

Take in a **MAILDIR_MSG** structure containing the filename and location of a message in a Maildir. Populate the other fields of the structure with info derived from the filename (i.e., the ’unique’ and ’info’ fields, and any other future subdivisions).

Arguments:

- **MAILDIR_MSG** * minfo: A structure representing a Maildir message

Return value:

int

0 for no error. -1 for error.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure

- maildir_status2info(): Build a Maildir filename ’info’ section

*** This function is for internal use only ***

Summary:

This function returns a string of an ’info’ filename segment corresponding to the given message status.

Arguments:

- int status: Bitmask of the current status

Return value:

char *

Proper ’info’ filename segment (Example: "2,RS" for replied-to-and-seen). NULL with **postal_errno()** set to POSTAL_E_NOERR if no info string necessary. NULL on error.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure

- mbox_alloc_spec(): Allocate a **SPEC_MBOX**

*** This function is for internal use only ***

Summary:

Allocate and initialize a **SPEC_MBOX** for use by the program.

Remember to use the **mbox_free_spec()** function to free the structure when you're done with it.

Arguments:

None.

Return value:

SPEC_MBOX *

Returns a pointer to an initialized **SPEC_MBOX** on success. Returns NULL on failure.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure

- mbox_derive_from(): Derive a From_ header

*** This function is for internal use only ***

Summary:

This function takes the header section of a mail message, and (if necessary) derives the mbox-style From_ header for it. It uses a number of reasonably intelligent heuristics to make the 'best guess' it can at what the header should be.

Note that the argument is a doubly-dereferenced pointer. This is necessary because adding a From_ header will increase the size of the string that stores the header, which means that it will (probably) need a new block of memory to hold the increased size. The old storage area will be **free()**'d internally.

Arguments:

- char ** header: An email header section

Return value:

int

0 on success. -1 on error.

Error codes:

- [POSTAL_E_INVAL]
  Bad arguments: Bad headers
- [POSTAL_E_NOMEM]
  Memory allocation failure

- mbox_free_spec(): Free a **SPEC_MBOX** structure

*** This function is for internal use only ***

Summary:

Takes a **SPEC_MBOX** structure and **free()**'s all its component parts.

Arguments:

- **SPEC_MBOX** * tofree: A single **SPEC_MBOX** structure.

Return value:

void

- mbox_lock_read(): Lock a mbox for reading

*** This function is for internal use only ***

Summary:

Locks the designated mbox file for reading (i.e., "shared").

Arguments:

- **POSTAL_CONN** * p_conn: An active mbox connection
- int locks: Bitmap of lock types to apply. See the description of **mbox_open_file()** for a description of possible values.

Return value:

int

Returns 0 if successful. Returns -1 if errors were encountered.

If an error is encountered, then any locks which did succeed were released before returning.

Error codes:

Note:

mbox_lock_read() may fail and set errors for any of the reasons listed in **postal_flock()**, **postal_dotlock()**, or **postal_fcntl_lock()**, depending on the locking mechanisms requested.

- mbox_lock_write(): Lock a mbox for writing

*** This function is for internal use only ***

Summary:

Locks the designated mbox file for writing (i.e., "exclusive").

Arguments:

- **POSTAL_CONN** * p_conn: An active mbox connection
- int locks: Bitmap of lock types to apply. See the description of **mbox_open_file()** for a description of possible values.

Return value:

int

Returns 0 if successful. Returns -1 if errors were encountered.

If an error is encountered, then any locks which did succeed were released before returning.

Error codes:

Note:

mbox_lock_write() may fail and set errors for any of the reasons listed in **postal_flock()**, **postal_dotlock()**, or **postal_fcntl_lock()**, depending on the locking mechanisms requested.

- mbox_set_conlen(): Set the Content-Length: header

*** This function is for internal use only ***

Summary:

Prepares the given message to be written into a mbox by setting (or correcting) the Content-Length: header.

Arguments:

- **POSTAL_MSG** * msg: A single mail message

Return value:

int

0 if successful. -1 on error.

Error codes:

Note:

mbox_set_conlen() may fail and set errors for any of the reasons listed in **postal_set_header()**.

- mbox_set_status(): Set the Status: and X-Status: headers

*** This function is for internal use only ***

Summary:

Updates the status information given in the headers of the message to match what's been set to be the new status in the **POSTAL_MSG** structure.

This is done from within **mbox_write_msg()** to bring things into line before the message is commited to disk. It should be used anytime a message with status information stored mbox-style in the headers is committed to stable storage.

Arguments:

- **POSTAL_MSG** * msg: A single mail message

Return value:

int

0 on success. -1 on error.

Error codes:

Note:

> **mbox_set_conlen()** may fail and set errors for any of the reasons listed in **postal_del_header()** or **postal_set_header()**.

- mbox_unlock(): Unlocks a mbox

*** This function is for internal use only ***

Summary:

Remove any and all locks applied to a mbox connection.

Arguments:

- **POSTAL_CONN** * p_conn: An active mbox connection

Return value:

int

Returns 0 on success. Returns -1 on failure to unlock.

If this returns an error, the locking may be in an indeterminate state. Bend over and kiss your ass goodbye.

Error codes:

Note:

> **mbox_unlock()** may fail and set errors for any of the reasons listed in **postal_flock()**, **postal_dotlock()**, **postal_fcntl_lock()**, depending on the locking mechanisms used.

- mbox_validate_from(): Validate a From_ line

*** This function is for internal use only ***

Summary:

Check a candidate From_ line with some simple heuristics to determine if it really IS a From_ line.

Arguments:

- const char * hdr: A pointer to the beginning of a candidate From_ line

Return value:

int

Returns 0 if From_ is valid (according to our heuristics). Returns -1 if invalid.

- pop_alloc_spec(): Allocate a **SPEC_POP**

*** This function is for internal use only ***

Summary:

Allocate and initialize a **SPEC_POP** for use by the program.

Remember to use the **pop_free_spec()** function to free the structure when you’re done with it.

Arguments:

None.

Return value:

SPEC_POP *

Returns a pointer to an initialized **SPEC_POP** on success. Returns NULL on failure.

Error codes:

- [POSTAL_E_NOMEM]
  Memory allocation failure

- pop_answer_check(): Checks command status

*** This function is for internal use only ***

Summary:

This function checks whether the response from the server to the previously sent command was affirmative or negative.

Arguments:

- **POSTAL_CONN** * server: An active POP connection

Return value:

int

Returns 0 if the response is affirmative ("OK"). Returns -1 if the answer is negative or error occured

Error codes:

- [POSTAL_E_SOCKET]
  Socket error: Unexpected EOF or read() error
- [POSTAL_E_SRVERR]
  Server error (read: "not-OK")

- pop_free_spec(): Free a **SPEC_POP** structure

*** This function is for internal use only ***

Summary:

Takes a **SPEC_POP** structure and **free()**'s all its component parts.

Arguments:

- **SPEC_POP** * tofree: A single **SPEC_POP** structure.

Return value:

void

- pop_network_close(): Fast-close a POP server connection

*** This function is for internal use only ***

Summary:

This closes a POP server connection in the quickest and dirtiest way possible. Never use it unless you're backed into a corner with error situations. **pop_close()** is a *MUCH* better way to close a connection.

This function will also free the memory used by the **POSTAL_CONN** structure you hand it.

Arguments:

- **POSTAL_CONN** * server: The active POP connection to close

Return value:

void