Messaging Access SDK Guide

Messaging Access SDK
C Version

Version 3.5


Recycled and Recyclable Paper

The Team: Messaging Access SDK Project Team, Manager: Gena Cunanan
Engineering: Soon Shin, Derek Tumulak, Prasad Yendluri
Marketing: Faten Hellal
Publications: Sharon Williams
Version 3.5

Part Number

Printed in the United States of America. 00 99 98 5 4 3 2 1

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

This guide describes the C version of the Netscape Messaging Access SDK 3.5, a development kit for writing messaging applications.

June 8, 1998

Contents

About This Guide

The *Messaging Access SDK Guide* is the developer's guide and reference to the Netscape Messaging Access software development kit (SDK), version 3.5, for writing messaging applications.

The *Messaging Access SDK Guide* tells you how to tap the capabilities of familiar and powerful Internet Access Protocols, POP3, IMAP4, SMTP, and MIME, in your messaging applications.

This guide describes the C version of the Messaging Access SDK.

Note For system requirements and installation information, see the `ReadMe` file that is available on the Netscape web site. §

This chapter includes the following sections:

- Who Should Read This Guide
- What's in This Guide
- Conventions Used in This Guide
- Where to Find More Information

[Top]

Who Should Read This Guide

The *Messaging Access SDK Guide* is designed for developers who want to create messaging applications based on the standard Internet protocols SMTP, POP3, MIME, and IMAP4.

[Top]

What's in This Guide

This is the guide to read if you want to write messaging applications using a variety of messaging access APIs. This guide comes in two versions, which document the C and Java versions of the Messaging Access SDK. You are reading the C version of the Guide.

- Organization
- Quick Reference to Tasks

[Top]

Organization

To provide quick access to conceptual information, task-based development information, and reference information, the guide consists of these parts:

- Using the Messaging Access SDK--Messaging Access SDK basics and development information.
- Messaging Access SDK C Reference--Reference to the C interfaces of the POP3, IMAP4, SMTP, and MIME APIs of the Messaging Access SDK.

[Top] [What's in This Guide]

Quick Reference to Tasks

To help you find the information you need more quickly, look in the column on the left for the task you want to perform. Click the title in the column on the right to go directly to the appropriate chapter.

If you want to do this:	See this chapter:
Learn more about Netscape Access APIs.	Chapter 1, "Introducing the Messaging Access SDK."
Understand how the Access APIs work together in a Messaging Access SDK application.	Chapter 1, "Introducing the Messaging Access SDK."
Send email messages.	Chapter 2, "Sending Mail with SMTP."
Encode, decode, and parse messages. Add attachments to messages.	Chapter 3, "Building and Parsing MIME Messages."
Retrieve manage messages on the server.	Chapter 4, "Receiving Mail with IMAP4,"
Retrieve messages, messages attributes, and parts of messages.	Chapter 4, "Receiving Mail with IMAP4."
Find out which Internet Protocol commands are called by Messaging Access SDK methods.	Chapter 7, "Reference to Protocols."
Find out what you need to know about using multithreading in your messaging applications.	Appendix A, "Writing Multithreaded Applications with the Messaging Access SDK."

To find the C functions and structures you need for your messaging application, see Part 2, "Messaging Access SDK C Reference."

[Top] [What's in This Guide]

Conventions Used in This Guide

Fonts. All program code listings, URLs, and other program names appear in Courier, a monospace font. Placeholders, which you replace with your own value, are in italicized Courier font.

Note Formats. This guide emphasizes information with several types of note formats:

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

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

Terminology. This guide uses the word *command* to represent Internet Protocol commands, and the word *function* to represent the Messaging Access SDK C implementation that calls this command. For example, the Messaging Access SDK `imap4_close` function sends the CLOSE IMAP4 protocol command.

[Top]

Where to Find More Information

For information for developers, see the Netscape DevEdge site.

This guide tells you how to use each Protocol API for Messaging Access SDK tasks. Internet Protocols are introduced and described in RFC (Request for Comments) documents from the Network Working Group. For further information about the Protocols, see the following RFCs.

SMTP RFCs

- RFC 821: “Simple Mail Transfer Protocol,” August 1982
- RFC 1854:”SMTP Service Extension for Command Pipelining,” October 1995
- RFC 1869: “SMTP Service Extensions,” November 1995
- RFC 1891: “SMTP Service Extension for Delivery Status Notifications,” January 1996
- RFC 2197: “SMTP Service Extension for Command Pipelining,” September 1997

MIME RFCs

- RFC 2045: “Multipurpose Internet Mail Extensions (MIME), Part One: Format of Internet Message Bodies,” November 1996
- RFC 2046: “Multipurpose Internet Mail Extensions (MIME), Part Two: Media Types,” November 1996
- RFC 2047: “MIME (Multipurpose Internet Mail Extensions), Part Three: Message Header Extensions for Non-ASCII Text,” November 1996
- For MIME headers: RFC 822: “Standard for the Format of ARPA Internet Text Messages,” August 1982

IMAP4 RFCs

- RFC 2060: “Internet Message Access Protocol - Version 4rev1,” December 1996
- RFC 2086: “IMAP4 ACL extension,” January 1997

- Internet Draft**:** “IMAP4 Namespace,” December 1997

POP3 RFCs

- RFC 1939: “Post Office Protocol - Version 3,” May 1996

Note An index to RFCs is available through the Internet FAQ Consortium. §

[Top]

1 *Using the Messaging Access SDK*

Chapter 5 Receiving Mail with POP3

This chapter is an overview of using POP3 (Post Office Protocol 3) to download messages to a client.

[Top]

Chapter

1

Introducing the Messaging Access SDK

The Netscape Messaging Access software development kit (SDK) provides a set of Protocol Level APIs that the developer can use to write messaging applications and extend applications with messaging services. This chapter is an overview of the Messaging Access SDK, version 3.5.

The Messaging Access SDK provides SMTP, MIME, POP3, and IMAP4 APIs in the Java and C programming languages, for a variety of platforms.

The *Messaging Access SDK Guide* provides developers with a complete set of software libraries, sample code, and documentation for building mail-enabled applications.

[Top]

How the Protocol APIs Work Together

The Messaging Access SDK provides implementations of the Internet messaging protocols, SMTP, IMAP4, MIME, and POP3. These Protocol APIs are designed to work together, yet have the ability to operate independently of each other.

SMTP (Simple Mail Transfer Protocol). SMTP sends plain text or MIME-encoded messages. You can use MIME to prepare to send messages in formats other formats than text, to encode messages, and to include attachments. For more information, see Chapter 2, “Sending Mail with SMTP.”

MIME (Multipurpose Internet Mail Extension). MIME builds and encodes messages with attachments for sending over SMTP, and parses and decodes received messages. The encoded MIME message is passed to SMTP.

The MIME API consists of the MIME encoder and the MIME parser. The MIME encoder is used to build MIME messages with attachments and encode them for sending over SMTP. You can use the MIME API to parse and decode messages when they are received through IMAP4 or POP3. For more information, see Chapter 3, “Building and Parsing MIME Messages.”

IMAP4 (Internet Message Access Protocol, version 4). IMAP4 is used to retrieve and manage messages remotely. The user can save messages on the server or locally. In addition, the user can manipulate items on the server (for example, create or delete mailboxes). IMAP4 supports multiuser mailboxes. For more information, see Chapter 4, “Receiving Mail with IMAP4.”

POP3 (Post Office Protocol, version 3). POP3 connects to the server and retrieves messages. POP3 is simpler than IMAP4 and provides a subset of its capabilities. It supports one user per mailbox. For more information, see Chapter 5, “Receiving Mail with POP3.”

For a quick reference to the Internet Protocol commands called by Messaging Access SDK methods, see Chapter 7, “Reference to Protocols.”

The Protocol APIs are designed to co-exist in the same client environment and match each other's interfaces where applicable. For example, the message data chunks returned by POP3 and IMAP4 APIs can be fed to the MIME SDK API to parse the message contents. In the same way, the encoded message byte-stream returned by the MIME API can be passed to the SMTP API to transmit the message. At the same time, the APIs are designed to allow customers to use only the API required by their application.

Applications written with the Messaging Access SDK Protocol APIs can work with any messaging system that implements the Internet messaging protocols, primarily the Netscape Messaging Servers. Protocol APIs are self-contained and are intended to coexist with all other SuiteSpot SDKs; they are independent of Netscape Server releases.

Each Messaging Access SDK Protocol API is designed to follow its Internet standard specification. API invocations result in the exchange of standard protocol elements with the server. Any information exchange with the server conforms to one of the standard protocols.

Figure 1.1 Messaging Access API Architecture

Messaging

The Messaging Access SDK is designed to parse and format protocol elements and make these available to the programmer through Java classes or C data structures and the methods and functions that access them.

The Messaging Access SDK Protocol APIs are built to be thin and are optimized for performance and memory. Each Protocol API includes a `sendCommand` (or pass-through) interface, which programmers can use to send protocol elements that are not directly supported by the API. The IMAP4 API has further conveniences, such as transparently handling unilateral and unsolicited responses from the server and making these available at the API level through a callback mechanism.

For more information about the `sendCommand` API, see the reference entries for `smtp_sendCommand`, `imap4_sendCommand`, and `pop3_sendCommand`.

For more information about using callbacks, see the section about callback mapping in each of these chapters: Sending Mail with SMTP, Receiving Mail with IMAP4, Receiving Mail with POP3, and Building and Parsing MIME Messages.

[Top]

The Messaging Access SDK, C Version

The Messaging Access SDK (Software Development Kit) comes in a zip file on Unix and a self-extracting executable on MS Windows.

You can download the SDK at this URL.

For the latest installation information, see the ReadMe file for the SDK.

The Messaging Access SDK download file contains the following directories and files:

- `include` directory
  -- C header files that supply common C definitions
  -- the `protocol` directory
- `include/protocol` directory - C header files for the Messaging Access SDK protocol implementations
- `lib` directory - Implementation library for each Protocol API
- `examples` directory - Sample code that demonstrates selected parts of the Messaging Access SDK
- `ReadMeC.htm` - Links to current installation information, development notes, system requirements, information about using the SDK that may be more current than this guide, and Netscape licensing information.

On Unix, unzip the downloaded file using a utility that preserves the file hierarchy, for example, gzip. On MS Windows, simply execute the self-extracting executable. For links to the latest information about installation, see `ReadMeC.htm`, included in the SDK.

[Top]

Supported Platforms

The Messaging Access SDK supports the MS Windows and Unix platforms listed in Table 1.1.

Table 1.1 Supported Platforms

Platforms	Supported Versions
Solaris	2.5.1, 2.6
Windows NT	4.0 with SP 3
Windows 95	
AIX	4.21
IRIX	6.2
DEC Unix	4.0d
HP-UX	11.0

[Top]

SDK Response Sinks for C

The SMTP, IMAP4, and POP3 response sinks and the MIME data sink are C structures made up of function prototypes and opaque data.

- The prototypes are patterns for the implementation of callbacks; they do not have implementations and do nothing in themselves.
- The opaque data is a pointer (`lptr`) that represents client data; this is set in the sink structure by the application. The sink structure is always returned to the application when a callback is are made.

For example, the SMTP response sink, `smtpSink_t`, defines opaque data and provides function prototypes for response functions.

```
typedef struct smtpSink
   /* User data. */

    void * pOpaqueData;

/* Notification for the response to the connection to the server. */

   void (*connect)( smtpSink_t * in_psmtpSink,
         int in_responseCode,
         const char * in_responseMessage );
   ...
} smtpSink_t;
```

Each function prototype is a callback that is associated with one or more functions in the SDK implementation of the protocol.

Functions with multi-line responses map to more than one callback, as shown in the table for `smtp_expand`. The second callback provides a notification that the operation is complete.

Function	Callback, Defined in `smtpSink_t`
`smtp_expand`	`void (*expand)(SMTPSinkPtr_t in_psmtpSink, int in_responseCode, const char * in_emailAddress );`
`smtp_expand`	`void (*expandComplete)(SMTPSinkPtr_t in_psmtpSink);`

For easy reference, each protocol chapter includes a table that shows how functions are mapped to callbacks. See the individual protocol chapters under “Function Callback Mapping.”

The developer implements the response sink by supplying the functionality for each method and associating each method implementation with its prototype. You can use any function name, but you must assign the same number and types of parameters. If you do not want to implement a function prototype, you can assign the function the value of `null`, as follows:

```
smtpSink_t.expand = null;
```

When you start a session with SMTP, IMAP4, or POP3, you first create (initialize) the sink. Then you create the client, which calls the sink functions. The sink receives and processes all the available server response data whenever `processResponses` is called. This call reads in responses from the server and invokes the appropriate callback function for all responses that are available at the time of execution.

You can create more than one response sink, based on what you want the messaging application to do. In SMTP, IMAP4, and POP3, you can change the response sink in use with the `setResponseSink` function.

When you start a session with the MIME dynamic parser, you first create and initialize the data sink, and then you create the parser. The parser makes callbacks to its data sink based on the kind of data it finds in the input stream. For example, if it finds a header, it makes the header callback. For the other protocols, the callback comes from the server and callbacks tend to be tied to individual functions.

For MIME, the kind of callback is dependent upon the kind of data that is in the input stream. There are no particular correspondences between functions and data sink callback prototypes, as there are in the other protocols.

For more information, see the section about implementing the response sink in each of these chapters: Sending Mail with SMTP, Receiving Mail with IMAP4, Receiving Mail with POP3, and Building and Parsing MIME Messages.

Note POP3, IMAP4, and POP3 commands are asynchronous. After sending a command, the application does not have to wait to issue the next one, but can do something else. §

[Top]

SDK Error Codes for C

The Messaging Access SDK Protocol implementations for SMTP, MIME, IMAP4, and POP3 return information about command operation using the set of standard SDK error codes defined in `nsmail.h`. For a complete list of SDK error codes, see Error Definitions.

Table 1.2 Messaging Access SDK Errors

Error Value	Description
`0`	`NSMAIL_OK`: Successful completion.
`<0`	Various error messages: Failure. Interpret the error and take appropriate action. See Error Definitions for the error types in this category.
`>0`	The intended action did not occur, but you can recover. For example, if the data is not returned or processed, you may still be able to go on processing.
`NSMAIL_ERR_IO_SOCKET`	Socket I/O error. As serious as other errors, but you can get more information about this type. On Unix, query the Unix `errno`. On Windows NT, call `GetlLastError()`.
`NSMAIL_ERR_TIMEOUT`	Time-out. Recoverable condition; the application can wait and reissue the function.

Note Error codes and other definitions to apply to all protocols are defined in Shared C Definitions. §

[Top]

Compiling with the C SDK

This section provides general information for including libraries and compiling with the SDK on the Unix and MS Windows platforms.

- Including C Header Files
- Compiling on Unix
- Compiling on MS Windows

[Top]

Including C Header Files

The protocol APIs in the Messaging Access SDK include libraries that implement the API functions. When you write messaging applications that use one or more of the SDK protocols, be sure to include the header files required by the APIs that you use.

When you install the Netscape C version of the Messaging Access SDK, the installation process creates an `include` directory and a `lib` directory under the `install` root.

- The `include` directory contains the files `nsstream.h` and `nsmail.h`. These are common `include` files for all of the Protocol APIs.
- The `include` directory contains a subdirectory named `protocol`. This subdirectory contains the individual protocol API-specific `include` files: `mime.h`, `mimeparser.h`, `smtp.h`, `pop3.h`, and `imap4.h`. The `mime.h` file contains definitions common to the MIME encoder and the MIME parser APIs. The `mimeparser.h` file contains definitions specific to the MIME parser API.

In your source files, include the Messaging Access SDK common header files `<nsmail.h>` and `<nsstream.h>`, along with any Protocol API include files: `<mime.h>` `<mimeparser.h>`, `<smtp.h>`, `<pop3.h>`, and `<imap4.h>`, that you need.

The `lib` directory contains the library files for the different protocol APIs: `libmime.*`, `libsmtp.*`, `libpop3.*`, and `libimap4.*`. The `lib` directory also contains the library that is common to all protocol APIs, `libcomm.*`. Here, the extension * represents the platform-specific extensions for the libraries: `.dll` or `.lib` on MS Windows; `.so` on Solaris and IRIX; `.sl` on HPUX; and `_shr.a` on AIX.

The `libmime` library contains the implementation of the MIME encoder as well as the MIME parser API. You must link with the `libmime` library if you intend to use the MIME encoder API, the MIME parser API, or both.

[Top] [Compiling with the C SDK]

Compiling on Unix

Follow these steps to compile the C version of the Messaging Access SDK on the Unix platform:

1. Copy the SDK API library files `libmime.*, libsmtp.*, libpop3.* , libimap4.* and libcomm.*` to your work directory, for example: `/usr/lib/msgsdk`
2. When compiling, be sure to include the library path in your link statement, for example: `-L/usr/lib/msgsdk -lmime -lsmtp -lcomm`
3. On Unix platforms, specify the `-DXP_UNIX` flag in your compilation (cc) step.
4. On the AIX platform only, also specify the -DAIX flag in your compilation step.
5. If you are using the POP3, SMTP, and IMAP4 protocols, you may also need to link with the additional libraries `-lsocket` and `-lnsl`.
6. When you run your client program, be sure to include the `msgsdk` library path in the `LD_LIBRARY_PATH` (on Solaris) or the equivalent on other platforms. This way, the clients can find the `msgsdk` shared libraries at run time.

[Top] [Compiling with the C SDK]

Compiling on MS Windows

Follow these steps to compile the C version of the Messaging Access SDK on MS Windows (NT/95) platforms:

1. On MS Windows platforms, you must link with the `libcomm.lib` and with one or more of the protocol-specific libraries `libmime.lib`, `libsmtp.lib`, `libpop3.lib`, and `libimap4.lib`.
2. Be sure to specify the directory where Messaging Access SDK `lib` files are located, for example, by using `/libpath` or by setting the `LIB` environment variable.
3. If required by your environment, specify the `-DWIN32` flag in your compilation (`cl`) step.
4. To ensure that the client you build with the Messaging Access SDK can find the dynamic link libraries, copy the `.dll` files to one of these locations:
   - The directory containing your client executable file.
   - The MS Windows `system` directory (Windows 95) or the `system32` directory (Windows NT).
   - Alternatively, you can specify the directory containing the dynamic link libraries in the `PATH` environment variable.

During run time, the Messaging Access SDK client looks for the dynamic link libraries in the following places:

- The current directory.
- The directory from which the application loaded.
- The 32-bit Windows `system` directory.
- The directories listed in the `PATH` environment variable.

[Top] [Compiling with the C SDK]

Chapter

2

Sending Mail with SMTP

This chapter is an overview of using SMTP (Simple Mail Transfer Protocol) to create and send email messages.

- The SMTP Protocol
- SMTP Function Callback Mapping
- Creating a Response Sink
- Creating a Client
- Connecting to a Server
- Determining ESMTP Support
- Setting the Mailer
- Setting the Recipient
- Sending the Message
- Ending the Session
- SMTP Functions by Task

[Top]

The SMTP Protocol

SMTP (Simple Mail Transport Protocol) allows clients to deliver mail messages to SMTP servers. To retrieve these messages, the client uses the IMAP4 or POP3 protocol. Servers can use SMTP to move messages from one server to another before delivering them to a mailbox.

The SMTP client always starts the session, but either client or server can end it. The client starts the session by connecting to the server. The server acknowledges the message with a greeting. The client responds, and, in subsequent commands, specifies the message sender and recipients and sends the message.

SMTP commands are made up of a keyword, followed by any parameters the function has. Commands receive a three-digit response code, described in SMTP Response Codes. SMTP commands include only the U.S. ASCII character set, a subset of ASCII that includes the values 00h-7Fh (0d-127d).

The responses returned by SMTP commands are made up of a three-digit numeric code followed by descriptive text. The client application can detect and handle the response or display the message to the user for interpretation. For more information, see SMTP Response Codes.

If your server supports Extended SMTP (ESMTP), which is provided in an update to the existing SMTP specification, your mail application can take advantage of ESMTP elements, such as pipelining, the `bdat` command, data chunking, and DSN. For more information, see Determining ESMTP Support.

During a single SMTP session, the client can send multiple unrelated, independently addressed messages. Because of this, the SMTP client can increase efficiency by batching messages and sending them together using pipelining. For more information, see Pipelining Commands.

The SMTP server waits for SMTP messages on the “well-known” TCP port 25. Many mail applications allow the user to specify a different port.

For a table of SDK-supported SMTP protocol commands, see Supported SMTP Internet Protocol Commands. For detailed information about SMTP, consult one of the RFCs listed, with links, in SMTP RFCs.

[Top]

Steps in an SMTP Session

Generally, a messaging application follows standard steps when using SMTP to send mail. These steps are listed below with links to more detailed descriptions.

Step	Section with details
Initialize the response sink.	Creating a Response Sink
Initialize the client.	Creating a Client
Connect to the server.	Connecting to a Server
Determine Extended SMTP (ESMTP) features supported by the server.	Determining ESMTP Support
Set the mailer.	Setting the Mailer
Set the recipients.	Setting the Recipient
Send the message.	Sending the Message
End the SMTP session.	Ending the Session

[Top]

SMTP Response Codes

When the client sends an SMTP command, the response that comes back contains a standard three-digit response code followed by descriptive text. This section is an overview of SMTP responses. For detailed information, see RFC 821.

The response contains the three-digit code, a space, and one or more lines of text that describes the response. If the response is multi-line, each subsequent line also contains the three digit code, a hyphen, and text. The final line contains the code, a space, and text. The three-digit response code and the rest of the text is made available through the callback functions. See `smtpSink_t`.

This table lists some of the most common SMTP reply codes. In general, response codes in the 100 to 300 range are considered successful; those in the 400 to 500 range are considered unsuccessful.

Table 2.1 SMTP Reply Codes

Code	Text of Response
211	system status, or system help reply
214	help message
220	<domain> service ready
221	<domain> service closing transmission channel
250	request mail action okay, completed
251	user not local, will forward to <forward-path >
354	start mail input; and with <CRLF>.<CRLF>
421	<domain> servers not available, closing transmission channel
450	requested mail action not taken: mailbox unavailable
451	requested action aborted: local error in processing
452	requested action not taken: insufficient system storage
500	syntax error, command unrecognized
501	syntax error in parameters or arguments
502	command not implemented
503	bad sequence of commands
504	command parameter not implemented
550	requested action not taken: mailbox unavailable
551	user not local; please try <forward-path>
552	requested mail action aborted: exceeded storage allocation
553	requested action not taken: mailbox name not allowed
554	transaction failed

The first digit of the SMTP reply code tells whether the response is positive or negative.

Table 2.2 SMTP Reply Codes, Digit 1

Digit 1	Meaning
1yz	Positive Preliminary Reply
2yz	Positive Completion Reply
3yz	Positive Intermediate Reply
4yz	Transient Negative Completion Reply
5yz	Permanent Negative Completion Reply

The information described by the second and third digits is noted here. For the meanings of specific numbers, see RFC 821.

- The second digit supplies response categories, such as Syntax or Connections, that identify the general type of failure.
- The third digit provides more information to help distinguish between responses with the same first two digits. For example, note the variations in the 55x codes in Table 2.1, SMTP Reply Codes. In all of these codes, the command failed, but for different reasons the code was able to identify, such as “mailbox unavailable” (550) or “unavailable or unable to find user” (551).

[Top]

SMTP Function Callback Mapping

Callbacks are associated with many SMTP functions. For general information about the response sink and callbacks, see SDK Response Sinks for C.

The smtpSink_t structure contains callbacks for each client call. The SMTP client's smtp_processResponses call invokes the corresponding interface function. This function reads in responses from the server and invokes the appropriate response sink functions. It thus invokes the callback functions provided by the user for all responses that are available at the time of execution.

If a time-out occurs, the user can continue by calling smtp_processResponses again.

Functions with multi-line responses map to more than one callback. The second callback provides a notification that the operation is complete.

If a server error occurs, the error callback is invoked.

Table 2.3 shows which SMTP functions are mapped to callbacks in the smtpSink_t structure. Table 2.4 shows which do not map to callbacks.

Table 2.3 Functions with Callbacks

Functions	Possible Responses, Mapped to Callbacks
smtp_bdat	bdat, error
smtp_connect	connect, error
smtp_data	data, error
smtp_ehlo	ehlo, ehloComplete, error
smtp_expand	expand, expandComplete, error
smtp_help	help, helpComplete, error
smtp_mailFrom	mailFrom, error
smtp_noop	noop, error
smtp_quit	quit, error
smtp_rcptTo	rcptTo, error
smtp_reset	reset, error
smtp_send	send, error
smtp_sendCommand	sendCommand, sendCommandComplete, error
smtp_sendStream	send, error
smtp_verify	verify, error

Table 2.4 Functions without Callbacks

Functions Without Callbacks
smtp_disconnect
smtp_free
smtp_get_option
smtp_initialize
smtp_processResponses
smtp_setChunkSize
smtp_setPipelining
smtp_setResponseSink
smtp_setTimeout
smtp_set_option
smtpSink_free
smtpSink_initialize

[Top] [SMTP Function Callback Mapping]

Creating a Response Sink

The first step in starting an SMTP session is to initialize the SMTP response sink, which is defined by the `smtpSink_t` structure. The response sink is made up of function pointers and opaque data. Initializing the sink sets its function pointers and opaque data to `null`. For general information about the response sink, see SDK Response Sinks for C.

To initialize and allocate the response sink, call the `smtpSink_initialize` function and supply the sink you want the SMTP client to use. If successful, this function returns `NSMAIL_OK`. Use this syntax:

```
int smtpSink_initialize( smtpSink_t ** out_ppsmtpSink );
```

The following section of code initializes the response sink and sets the sink function pointer.

```
int l_retCode = 0;

l_retCode = smtpSink_initialize(&l_psmtpSink);

if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }

setSinkFunctions(l_psmtpSink);
```

After you create the response sink, the next step is Creating a Client.

When a session is finished, you must free any memory associated with the response sink. To free the response sink and its data members, use `smtpSink_free`:

```
void smtpSink_free( smtpSink_t ** in_ppsmtpSink );
```

When this function returns, the response sink is set to `null`. The user must free any opaque data.

[Top] [Creating a Response Sink]

Creating a Client

The SMTP client uses the `smtpClient_t` structure to communicate with an SMTP server. To initialize and allocate the `smtpClient_t` structure and set the response sink for the client's use, call the `smtp_initialize` function.

When you create the client structure, you supply the address of the pointer to the SMTP client you are creating, along with an initialized response sink, as described in Creating a Response Sink. Use this syntax:

```
int smtp_initialize( smtpClient_t ** out_ppSMTP,
                     smtpSink_t * in_psmtpSink );
```

The following section of code creates a client.

```
/* Initialize sink first as described in Creating a Response Sink */

l_retCode = smtp_initialize(&l_psmtpClient, l_psmtpSink);

if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
```

When a session is finished, you must free any memory associated with the client. To free the client structure and its data members, use this function:

```
void smtp_free( smtpClient_t ** in_psmtp_Client );
```

The `in_psmtp_Client` parameter represents the SMTP client. When this function returns, the client structure is set to `null`.

After you initialize the client, the next step is Connecting to a Server.

[Top]

Connecting to a Server

Before sending mail, the client must connect with the server through a service port. To connect to the server, use the `smtp_connect` function.

This function requires the identifier of the SMTP client that wants to connect with the server. If the server is using the default port for the SMTP protocol (port 25), you can pass 0 as the value of the port parameter. Use this syntax:

```
int smtp_connect(smtp_t * in_pSMTP,
                 const char * in_server,
                 unsigned short in_port );
```

On connecting, the server sends a greeting message to the client. The client responds by identifying itself with the EHLO command.

Note For this function's callback mapping, see SMTP Function Callback Mapping. §

The following section of code connects the client to the server.

```
/* After Creating a Response Sink and Creating a Client */

int l_retCode = 0;

char* l_szServer = "smtpServer.com";

l_retCode = smtp_connect(l_psmtpClient, l_szServer, 25);

if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }

l_retCode = smtp_processResponses(l_psmtpClient);

if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
```

During the connect process, you can enable pipelining if your server supports it. See Pipelining Commands. To find out which extensions are supported by the server, see Determining ESMTP Support.

After connecting to the server, the next step is Setting the Mailer.

When a session is finished, you must free any memory associated with the client. To disconnect the client from the server and close the socket connection, use this function.

```
int smtp_disconnect( smtpClient_t * in_pSMTP );
```

The `in_pSMTP` parameter represents the SMTP client. You can use this function as part of a Cancel operation while retrieving a message. Remember that you do not call `smtp_processResponses` after `smtp_disconnect`.

[Top]

Determining ESMTP Support

To retrieve a listing of extensions that are supported by the server, call the `smtp_ehlo` function. This function returns a multiline message listing the Extended SMTP (ESMTP) features, such as pipelining or DSN, that the server supports. This is similar to the functionality of the IMAP4 `capability` command. Use this syntax:

```
int smtp_ehlo( smtpClient_t * in_pSMTP,
               const char * in_domain );
```

This function calls the EHLO SMTP protocol command, which can be issued in any session state, but is usually issued after connecting to the server.

Note For this function's callback mapping, see SMTP Function Callback Mapping. §

The following section of code finds out which Extended SMTP (ESMTP) features the server supports.

```
/* After Connecting to a Server */

int l_retCode = 0;

l_retCode = smtp_ehlo(l_psmtp, l_szDomainName);
```

```
if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
l_retCode = smtp_processResponses(l_psmtpClient);
if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
```

You can enable pipelining if your server supports this extension. See Pipelining Commands.

[Top]

Pipelining Commands

Pipelining allows you to group, or batch, functions for execution rather than executing each separately. If pipelining is enabled on your server, commands are stored internally in the client as they are issued. All commands begin to execute when triggered in one of three ways: if the `smtp processResponses` function is called, if the internal storage area is full, or if a function that cannot be pipelined is issued.

You can enable pipelining anywhere, but it may make sense to do this after invoking the `smtp ehlo` function. Pipelining may then be enabled if the server supports it. If not, the way the network works does not change. The `ehlo` callback indicates whether pipelining is supported.

Use this syntax to attempt to enable pipelining:

```
int smtp_setPipelining( smtpClient_t * in_pSMTP,
                        boolean in_enablePipelining );
```

The `in_pSMTP` parameter represents the client. The `in_enablePipelining` parameter is a Boolean value that tells the server to attempt to enable pipelining. This function sends the PIPELINING SMTP protocol command.

Note For this function's callback mapping, see SMTP Function Callback Mapping. §

Some functions continue to add to the pipelining list. These are `smtp_bdat`, `smtp_mailFrom`, `smtp_rcptTo`, and `smtp_sendStream`. Calling any other function causes the commands on the pipelining list to be sent.

For example, you could call `smtp_mailFrom`, followed by one or more calls to `smtp_rcptTo`. These functions are added to the pipelining list and are not executed. If you then call another function, such as `smtp_noop`, the commands are sent to the server.

For details about using pipelining, refer to RFC 1854, "SMTP Service Extension for Command Pipelining."

[Top]

Setting the Mailer

Setting the mailer starts the process of delivering a message. Supply the identifier of the SMTP client that is sending the mail. If your server supports Extended SMTP, you can pass ESMTP elements in the `in_esmtpParams` parameter. Use this function:

```
int smtp_mailFrom(smtpClient_t * in_pSMTP,
                  const char * in_reverseAddress,
                  const char * in_esmtpParams );
```

The function identifies the sender and provides the sender's fully qualified domain name in the `in_reverseAddress` parameter. It sends the MAIL FROM: SMTP protocol command.

Note For this function's callback mapping, see SMTP Function Callback Mapping. §

The following section of code sets the mailer.

```
/* After Connecting to a Server */

int l_retCode = 0;

char* l_szServer = "smtpServer.com";

char* l_szDomainName = "netscape.com";

char* l_szMailer = "derekt@netscape.com";

l_retCode = smtp_mailFrom(l_psmtp, l_szMailer, NULL);

if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }

l_retCode = smtp_processResponses(l_psmtpClient);
```

```
if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
```

After you set the mailer, the next step is Setting the Recipient.

[Top]

Setting the Recipient

After setting the mailer, the next step is to set the recipient. Use this function:

```
int smtp_rcptTo(smtpClient_t * in_pSMTP,
                const char * in_forwardAddress,
                const char * in_esmtpParams );
```

This function sets a single recipient, so you must call it again for each recipient of a message. The `in_pSMTP` parameter represents the client. The `in_forwardAddress` parameter supplies the recipient's address. If your server supports Extended SMTP, you can pass ESMTP elements in the `in_esmtpParams` parameter.

`smtp_ rcptTo` sends the RCPT TO SMTP protocol command.

Note For this function's callback mapping, see SMTP Function Callback Mapping. §

The following section of code sets the recipient.

```
/* After Setting the Mailer */

int l_retCode = 0;

char* l_szRecipient = "alterego@netscape.com";

smtpClient_t * l_psmtpClient = NULL;

l_retCode = smtp_rcptTo(l_psmtp, l_szRecipient, NULL);

if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }

l_retCode = smtp_processResponses(l_psmtpClient);

if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
```

After you set the recipient, the next step is Sending the Message.

[Top]

Sending the Message

After setting all of the message recipients, the client can send the message data. To send a message, use `smtp_data`, followed by either the `smtp_send` or the `smtp_sendStream` function. First, use this function:

```
int smtp_data( smtpClient_t * in_pSMTP );
```

Supply the identifier of the SMTP client that is sending the mail. The server responds with a success or failure reply code. See SMTP Response Codes.

The `smtp_send` and `smtp_sendStream` functions both deliver data to the server. `smtp_send` sends data in a single chunk, while `smtp_sendStream` sends the data in a series of smaller chunks. If you use either of these functions, you must send data with the `smtp_data` function and not with `smtp_bdat`.

`smtp_data` and `smtp_bdat` are interchangeable, but cannot be used together. The `smtp_bdat` function, which can deliver binary data, is not supported on the Netscape Messaging Server and some other servers.

After the `data` function, call `smtp_send`, which sends a message to the server:

```
int smtp_send( smtpClient_t * in_pSMTP, const char * in_data );
```

Supply the identifier of the SMTP client that is sending the mail and the identifier for the data to send. When the server responds that it is ready, the client sends the RFC 822 message data line by line.

You can set data chunk size with `smtp_setChunkSize`, or you can use the default (1 K). You can set this to specify the chunk sizes used by the `smtp_sendStream` function.

The `smtp_sendStream` function, like `smtp_send`, sends a message to the server.

```
int smtp_sendStream( smtpClient_t * in_pSMTP,
                     nsmail_inputstream_t * in_inputStream );
```

Supply the identifier of the SMTP client that is sending the mail and the identifier for the input stream.

Note For the callback mapping for these functions, see SMTP Function Callback Mapping. §

The following section of code uses `smtp_data` and `smtp_send` to send a message.

```
smtpClient_t * l_psmtpClient = NULL;
/*Sending the Message*/
/* Start by sending identifier of sender */
l_retCode = smtp_data(l_psmtpClient);
if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
l_retCode = smtp_processResponses(l_psmtpClient);
if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
/*Send the Message with smtp_send */
l_retCode = smtp_send(l_psmtpClient, "Hi how are you today?");
if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
l_retCode = smtp_processResponses(l_psmtpClient);
if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
```

After you send the message and perform any other SMTP operations you need for the session, the next step is Ending the Session.

[Top]

Ending the Session

When the client wants to close the session, the messaging application must free the response sink, free the client, and close the socket connection with the server. For these operations, see Connecting to a Server, Creating a Client, and Connecting to a Server.

The client calls `smtp_quit` to notify the server. The server closes the TCP connection and returns a response code. It is preferable to end a session with `smtp_quit` instead of just closing the connection. This function sends the QUIT SMTP protocol command:

```
int smtp_quit( smtpClient_t * in_pSMTP );
```

Supply the identifier of the SMTP client that wants to end the session.

Note For this function's callback mapping, see SMTP Function Callback Mapping. §

The following section of code notifies the server that the client is terminating the session.

```
int l_retCode = 0;
smtpClient_t * l_psmtpClient = NULL;
l_retCode = smtp_quit(l_psmtp);
if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
l_retCode = smtp_processResponses(l_psmtpClient);
if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
```

[Top]

SMTP Functions by Task

To help you find the information you need more quickly, look for the task you want to perform in the column on the left. Then you can click the function name for further information.

Table 2.5 SMTP Functions by Task

Task	Function for the task
Deliver data in binary data chunks to the server.	smtp_bdat
Connect to the server using the default port.	smtp_connect
Deliver data to the server.	smtp_data
Close the socket connection.	smtp_disconnect
Send the EHLO command to the server.	smtp_ehlo
Expand a mailing list.	smtp_expand
Free the SMTP client structure and its data members.	smtp_free
Get the IO models.	smtp_get_option

Task	Function for the task
Asks the server for help on a specified topic.	smtp_help
Initialize and allocate the smtpClient_t structure and set the sink.	smtp_initialize
Initiate sending message; supplies the message's reverse path.	smtp_mailFrom
Get a positive server response without affecting the SMTP session.	smtp_noop
Process the server responses for API commands.	smtp_processResponses
Close the connection with the server.	smtp_quit
Specify a message recipient's address.	smtp_rcptTo
Reset the state of the server and discard any sender and recipient information.	smtp_reset
Send message data to the server.	smtp_send
Send an unsupported command to the server.	smtp_sendCommand
Sends an SMTP command that is not otherwise supported by this API.	smtp_sendStream
Set the size of data chunks to send.	smtp_setChunkSize
Set IO and threading models.	smtp_set_option
Attempt to enable pipelining.	smtp_setPipelining
Set a new response sink.	smtp_setResponseSink
Set the amount of time allowed to wait before returning control to the user.	smtp_setTimeout
Verify a username.	smtp_verify
Free the SMTP response sink and its data members.	smtpSink_free
Initialize and allocate the SMTP response sink structure.	smtpSink_initialize

[Top] [SMTP Functions by Task]

Chapter

3

Building and Parsing MIME Messages

This chapter is an overview of using the MIME (Multipurpose Internet Mail Extension) API of the Messaging Access SDK to encode, decode, and parse mail messages, and handle text and non-text attachments.

- The MIME Protocol
- Structure of a MIME Message
- Steps in a MIME Session
- Building the MIME Message
- Encoding the Message
- Parsing MIME Messages
- MIME Functions by Task

[Top]

The MIME Protocol

The MIME (Multipurpose Internet Mail Extension) protocol is the solution for sending multipart, multimedia, and binary data over the Internet. MIME is the standard for sending a variety of data types, including video, audio, images, programs, formatted documents, and text, in email messages.

The MIME protocol is made up of the extensions to the Internet mail format documented in RFC 822, "Standard for the Format of ARPA Internet Text Messages," August 1982. The MIME protocol, documented in a series of MIME RFCs, adds these features:

- The ability to send rich information through the Internet
- The ability to encode and attach binary (non-ASCII) content to messages
- A framework for multipart mail messages that contain differing body parts
- A way to identify the content type associated with a message body part
- A standardized and interpretable set of body part types

MIME messages can include attachments and non-ASCII data. To conform with RFC 822, which requires mail message characters to be in ASCII, MIME uses an encoding algorithm to convert binary data to ASCII characters. For content that requires encoding, MIME specifies two encoding types, either Quoted-Printable or BASE64, which are described more fully in MIME Encoding Types.

The MIME API of the Messaging Access SDK provides the ability to build multimedia messages in MIME format plus a parsing facility for messages. The MIME parser takes a MIME-encoded email message and decodes all or parts of it, depending on the preferences of the application. The MIME parser is described in Parsing MIME Messages.

For detailed information about MIME, consult one of the RFCs listed, with links, in MIME RFCs.

[Top]

MIME Encoding Types

MIME messages can include attachments and non-ASCII data. RFC 822 requires mail message characters to be in ASCII, so MIME uses an encoding algorithm to convert binary data to ASCII characters. MIME uses one of two encoding types, Quoted-Printable and BASE64 encoding.

Quoted printable encoding handles content that is mostly composed of ASCII characters, with only a small number that are non-ASCII (for example, Scandinavian characters in the ISO-8859-1 character set). This text is mostly readable on the client before it is encoded. The encoding process ignores ASCII characters and encodes the rest, using a set of rules for representing characters, line breaks, and tabs, and limiting line length.

BASE64 encoding handles binary data. This algorithm works by encoding sets of a octets into encoded characters, and produces 33 percent data expansion.

The MIME API also supports a non-encoding option. For example, no encoding is required for text messages.

For detailed information about MIME encoding, consult one of the RFCs listed, with links, in MIME RFCs.

[Top]

MIME Content Types

MIME types typically have three parts, a type, a subtype and optional content-type parameters. The type is the general content category; the subtype is the specific data format, as shown in these examples:

- `text/plain`: Text content (type) in `plain` format (subtype).
- `image/gif`: An `image` file (type) in `gif` format (subtype).

This table lists the valid MIME content types. A valid subtype can be of any data format type, including numerous experimental formats.

Table 3.1 MIME Content Types

Type	Description	Subtypes
Text	Information in raw text form. Has optional character set (default: `us-ascii`).	plain: includes no formatting information
Audio	Message body contains audio data.	basic
Image	Message body contains an image.	image format name, for example: `gif, jpeg`
Video	Message body contains a time-varying-picture image, possibly with color and sound.	image format name, for example: `mpeg`
Application	Uninterpreted binary data or information to be processed by an application.	octet-stream, postscript
Multipart	Messages with multiple attachments of potentially different media. Subtypes describe how the sub-parts relate.	mixed, alternative, digest, parallel
Message	Identifies a message.	rfc822, partial, external-body

The MIME implementation of the Messaging Access SDK provides functions that create these content types, add them to messages, and encode or decode them.

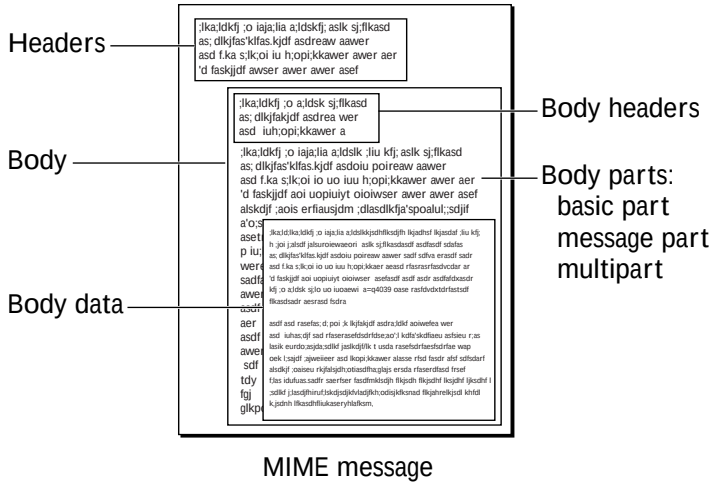
For detailed information about MIME content types, subtypes, and content parameters, consult one of the RFCs listed, with links, in MIME RFCs.

[Top]

Structure of a MIME Message

A MIME message has two main parts, the header and the body.

Figure 3.1 Parts of a MIME message



The message header consists of lines that describe the sender, subject, recipient, date, version of MIME in use, and a variety of other types of information, depending on the needs of the messaging application. This example shows the header lines of a message.

```
Return-Path:<Prasad@netscape.com>
   Received:from netscape.com ([205.217.229.85])by
            dredd.mcom.com (Netscape Messaging Server
            3.0) with ESMTP id AAA24896; Wed, 4 Feb 1998
            20:08:19 -0800
     Sender:prasad
 Message-ID:<34D93795.C1F48C83@netscape.com>
       Date:Wed, 04 Feb 1998 19:52:53 -0800
       From:Prasad Yendluri <Prasad@netscape.com>
  X-Mailer:Mozilla 4.03C-NSCP [en] (X11; U; SunOS 5.5.1
            sun4u)
MIME-Version:1.0
         To:sharonw@netscape.com
    Subject:Information about MIME
Content-Type:multipart/mixed; boundary =
            "------------ BFA9E722569728E3111F0326"
```

For more information, see Adding Message Headers.

The message body consists of body parts of different types, depending on the demands of the data in the message.

- Basic part. Includes all Basic MIME body part types: text, image, audio, video, and application. It does not include the Message-part or Multipart types. This is the simplest part.
- Multipart. Container part made up of two or more sub-body parts. The Multipart type includes several subtypes that describe how the sub-parts relate to each other (mixed, alternative, digest, and parallel).
- Message part. Message used as an attachment, for example, a message forwarded in another message.

For more information, see Building the MIME Message.

[Top]

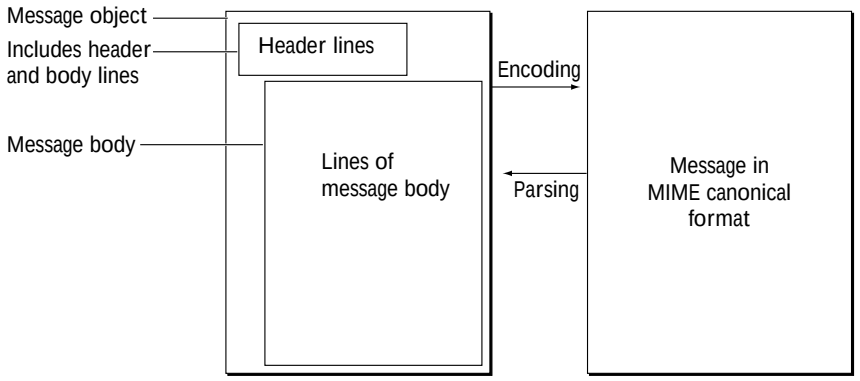
Steps in a MIME Session

The basic MIME operations focus on preparing a message to be sent and translating a received message into readable form for a mail application. Before the MIME message can be sent, the message and its attachments must be built and encoded in MIME format. When a message is received, it must be parsed and decoded.

Generally, a messaging application follows these steps when using MIME to build a message.

- Building the MIME Message.
- Encoding the Message.
- Decoding and parsing the Message. See Parsing MIME Messages.

Figure 3.2 Translating a MIME Message



[Top]

Building the MIME Message

Before sending a MIME message, you must build the message and its attachments and encode them in MIME format. This section describes the steps in building a message.

You can build MIME messages from RFC headers and simple text or attachments or both. You have the option of either building the message part by part, or using one of the convenience functions provided by the API.

There are two stages in building a message.

First, create an empty message structure. This example creates a pointer to a message called `pNewMessage`.

```
mime_message_t *pNewMimeMessage =
   (mime_message_t *) mime_malloc (sizeof (mime_message_t));
```

Then, add two components, in either order.

- Add the message headers. Headers are `name:value` pairs that conform to the requirements of RFC 822. For more information, see Adding Message Headers.
- Add the content. Content can simply be text or it can include several parts or file attachments. For more information, see Adding Content to the Message.

You can also create a message with `mime_message_create`, a convenience function that builds a MIME message structure:

```
int mime_message_create (char * in_pText,
                         char * in_pFileFullName,
                         mime_encoding_type in_perf_file_encoding,
                         mime_message_t ** out_ppMessage);
```

You provide the filename of an attachment or text; one of these is required. Also, supply the file encoding type to use (either Base64, Quoted Printable, or -1 to use a default). This function creates a message, to which you add the headers.

One of the sample applications in the `examples` directory of the SDK illustrates building a message using this function.

The following section of code demonstrates using `mime_message_create` to build a MIME message with text, a data buffer, and a file.

```
/* Build a MIME Message with "text" and the file “/tmp/testfile.txt” */

ret = mime_message_create (pBuf, “/tmp/testfile.txt”, enc, &pMsg);

/* Build a message with just text and NO file. */

ret = mime_message_create (pBuf, NULL, 0, &pMsg);

/* Build a message with just a file and NO text. */

ret = mime_message_create (NULL, “/tmp/testfile.txt”, 0, &pMsg);
```

Next, add the headers to the message. See Adding Message Headers.

[Top]

Adding Message Headers

After building a message, add the message headers. To do this, use the `mime_header_new` function. You supply the header name and value. This function creates a header entry as a `name:value` pair and adds it to the message.

```
mime_header_t * pHdr = mime_Header_new (
                char * in_pName, char * in_pValue);
```

To create a header, this function populates a `mime_header_t` structure with the name and value parameters you specify. Now you can add this header to the RFC 822 header part of the MIME message.

The following section of code creates RFC 822-compliant headers for a MIME message.

```
mime_header_new ("From", "prasad@netscape.com");

mime_header_new ("To", "smith@netscape.com");

mime_header_new ("Subject", "Using MIME");

mime_header_new ("X-Msg-SDK-HDR", "X-Test-Value1");
```

When the headers are created, add them to the `rfc822_headers` member of the MIME message structure `mime_messagePart_t`.

Next, you can add any of the other message parts or attachments that the message requires. See Adding Content to the Message.

[Top]

Adding Content to the Message

To add content to a message, you first create a message body part, then use MIME API functions to add the data to the part. You can then add this part, which now contains data, to the message. A MIME message can include the following types of body parts:

- Basic part. The simplest basic part is text that you type in. Other basic parts are audio, video, image, or application files.

- Multipart. A multipart contains two or more basic parts. The constructed multipart is added to the message. Use this content type if you are constructing a more complex message.
- Message part. A message part is a message that becomes an attachment, for example, when it is forwarded.

Creating the Basic Part and Adding Data

Before you can add a basic part, you first create the basic part structure, `mime_basicPart_t`. This is the common structure for the leaf parts, text, audio, video, image, and application.

Create the basic part by defining the `mime_basicPart_t` structure and setting up the attributes of the part, including elements such as content description, content type, and headers, including X-headers.

After you set up the basic part structure and its attributes, you can add the body data to the structure. To do this, use either `mime_basicPart_setDataBuf` or `mime_basicPart_setDataStream`, depending on the source of the data.

```
int mime_basicPart_setDataBuf (
               mime_basicPart_t * in_pBasicPart,
               unsigned int in_size,
               const char * in_pDataBuf,
               BOOLEAN in_fCopyData);
```

The `in_pBasicPart` parameter identifies the basic part that you are adding the data to; you must already have created this part. The `in_pDataBuf` parameter represents the data source. Use the `in_fCopyData` parameter to specify whether to the function should copy the data or keep a reference to the data buffer.

Repeat this for each basic part you add to the multipart.

The `mime_basicPart_setDataStream` function works in the same way, except that the data source is an input stream.

```
int mime_basicPart_setDataStream (
               mime_basicPart_t * in_pBasicPart,
               mime_inputstream_t * in_ptheDataStream,
               BOOLEAN in_fCopyData);
```

Now you can add this part to the message. See Adding Parts to the Message.

[Top]

Building a Multipart

If a message has two or more attachments, you must create a multipart that includes them before you can add them the message.

- First, create each part as a basic part. For more information, see Creating the Basic Part and Adding Data.
- Then create a multipart and add each basic part to it.
- After this, you can add this multipart to the message.

This example demonstrates creating a multipart.

```
mime_multiPart_t * pMultiPart = (mime_multiPart_t *)
               mime_malloc (sizeof (mime_multiPart _t);
```

To add basic parts to a multipart, use one of these MIME API functions:

- mime_multiPart_addBasicPart. Adds an existing basic part to a multipart. See Adding a Basic Part to a Multipart.
- mime_multiPart_addMessagePart. Adds an existing message part to a multipart.
- mime_multiPart_addMultiPart. Adds an existing multipart to a multipart.
- mime_multiPart_addFile. A convenience function that adds the specified file as a basic part to a multipart.

Adding a Basic Part to a Multipart

Before you can add a basic part to a multipart, you must create the basic part structure and add data to it. See Creating the Basic Part and Adding Data.

When the basic part is complete, you can it to the multipart. To do this, use mime_multiPart_addBasicPart.

```
int mime_multiPart_addBasicPart (
               mime_multiPart_t * in_pMultiPart,
               mime_basicPart_t * in_pBasicPart,
```

```
                BOOLEAN in_fClone,
                int * out_index_assigned);
```

You supply the existing basic part that you are adding, the multipart to add it to, and the index at which you want to add it. Use the `in_fClone` parameter to specify whether to the function should keep a reference to the body part object itself or a cloned copy. Repeat this for each basic part you add to the multipart.

After creating and assembling the multipart, the next step is to add the multipart to the message. See Adding Parts to the Message.

[Top]

Adding Parts to the Message

After you create the MIME message structure and the body parts you want it to include, use the MIME Message function group to add the parts to the message. Different functions add basic parts, multiparts, and message parts.

- `mime_message_addBasicPart`. Adds a basic part to a message.
- `mime_message_addMessagePart`. Adds a message part to a message.
- `mime_message_addMultiPart`. Adds a multipart to a message.

For example, when the basic part is complete and includes data, as described in Creating the Basic Part and Adding Data, add it to the message with the `mime_message_addBasicPart` function, shown here:

```
int mime_message_addBasicPart (
                mime_message_t * in_pMessage,
                mime_basicPart_t * in_pBasicPart,
                BOOLEAN in_fClone);
```

The basic part becomes the body of the message. The content-type of the message is set according to the content type of the basic part. If the message already has a message body, an error occurs. Use the `in_fClone` parameter to specify whether to the function should keep a reference to the body part object itself or a cloned copy.

You can simplify the process of building and adding content to a message by using the convenience function `mime_message_create`. You supply text, a file to attach, and the encoding type. The function creates a message, to which you add the headers. For more information about this function, see Building the MIME Message.

[Top]

Adding a Message Part to a Message

When you forward a message, it becomes the content of another message. This means that the mail application must do two things:

- Make a message part from the message structure to be forwarded.
- Add it as the content (body) of the message to be sent.

To make a message part from the message structure, use this function:

```
int mime_messagePart_fromMessage (
               mime_message_t * in_pMessage,
               mime_messagePart_t ** out_ppMessagePart()
```

This function takes the message structure and returns it as a message part.

To add the message part as the body of the message, use this function.

```
int mime_message_addMessagePart (
               mime_message_t * in_pMessage,
               mime_messagePart_t * in_pMessagePart,
               BOOLEAN in_fClone);
```

This function fails if a message body already exists. The `clone` parameter should contain `true` if the function should clone a copy of the message or `false` if it should store a reference to the passed object.

[Top]

Deleting Parts of a Message

If you want to delete the message, a body part (or parts of it) after it is built, use one of the MIME delete functions. You can delete the entire message, the body, or a message part, as needed.

- `mime_basicPart_deleteData`. Deletes the body data for a part.
- `mime_multiPart_deletePart`. Deletes a body part from a multipart.
- `mime_messagePart_deleteMessage`. Deletes a MIME message that is the body of a message part.
- `mime_message_deleteBody`. Deletes the body of a message.

[Top]

Encoding the Message

After building a message, the next step is to encode it. You can encode an entire message, with headers and message attachments, in one operation with the `mime_message_putByteStream` function. Encoding puts messages into MIME canonical form, so that they can be transmitted over SMTP and other transport functions.

```
int mime_message_putByteStream (
        mime_message_t * in_pMessagePart,
        mime_outputstream_t * in_pOutput_stream);
```

The function required pointers to the message that contains data to encode and to the MIME output stream for the encoded data. For the SDK functions that create input and output streams, see Shared Functions.

After the message is encoded, you can send it using SMTP. For information about SMTP, see Chapter 2, “Sending Mail with SMTP.”

The following section of code writes the MIME-encoded message to the specified target, in this case, a file.

```
/* Writes the MIME encoded message to a file */

sprintf (filename, "%s%s", TMPDIR, "sdkCEnc.out");

file_outputStream_create (filename, &pOS);

mime_message_putByteStream (pMsg, pOS);

file_outputStream_close (pOS->rock);

nsStream_free (pOS);
```

[Top]

Encoding and Decoding Utilities

The MIME API provides a number of utility functions for encoding and decoding. These functions are used by other MIME API functions internally, and are also made available to developers to use in their applications.

The `mime_message_putByteStream` function could call either of two encoding utility functions, `mime_encodeBase64` or `mime_encodeQP`, based on whether the requested encoding type is Base64 (default for non-text types) or Quoted Printable. These utility functions each provide a single form of encoding. Like `mime_message_putByteStream`, these functions take an input stream and encode it. If an application requires only Base64 or QP-encoded data, you can use one of these functions in place of `mime_message_putByteStream`.

- `mime_encodeBase64`. Base64 encodes data and writes it to an output stream.
- `mime_encodeQP`. Quoted Printable encodes the data from an input stream and writes to output stream.

The `parseEntireMessage` and `parseEntireMessageInputstream` functions, which parse and decode encoded messages, could call either of two decoding utility methods, `mime_decodeBase64` or `mime_decodeQP`, based on the requested encoding type. These utility methods each provide a single form of decoding, and could be used instead of `parseEntireMessage` if an application only needs to decode Base64 or QP-encoded data.

- `mime_decodeBase64`. Base64 decodes the data from an input stream and writes to an output stream.
- `mime_decodeQP`. Quoted Printable decodes the data from an input stream and writes to an output stream.

[Top]

Encoding and Decoding Headers

Two utility functions allow you to encode and decode only the headers of a message.

- `mime_encodeHeaderString`. Encodes an RFC 2047-compliant header from an input stream, using Base64 or Q encoding. You can select the character set for the input stream. The header string can be used as the value of unstructured headers or in the comments section of structured headers.
- `mime_decodeHeaderString`. Decodes only the RFC 2047-compliant header of a message.

[Top]

Parsing MIME Messages

For parsing encoded messages, the Messaging Access SDK provides these options:

- Parsing the Entire Message. Use this option when the message to be parsed is available in its entirety when you begin parsing.

- Dynamic Parsing. Use the dynamic parser when the entire message is not available when you begin parsing, but becomes available block by block. This could happen when you are receiving a message from a server.

[Top]

Parsing the Entire Message

You can use MIME functions to parse and decode encoded messages retrieved through email protocol APIs, such as POP3 and IMAP4.

The MIME API of the Messaging Access SDK provides two functions for parsing an entire MIME message in one operation. These functions either parse a message that comes from a data buffer or from an input stream. Both return a MIME Message structure that represents the parsed message. They are defined in `mimeparser.h`.

To parse a message from a data buffer, use `parseEntireMessage`:

```
mime_message_t * parseEntireMessage (char *pData,
                int nLen,
                struct mime_message ** ppMimeMessage);
```

Supply a pointer to the message data and the length of the data. The function returns the parsed message.

To parse a message from an input stream, use `parseEntireMessageInputstream`:

```
int parseEntireMessageInputstream(
                struct nsmail_inputstream *pInput,
                struct mime_message **ppMimeMessage);
```

Supply the identifier for the stream that is the source of the data and the destination of the message. For the SDK functions that create input and output streams, see Shared Functions.

The following section of code uses `parseEntireMessageInputstream`, which parses the encoded message from an input stream, as part of a routine that parses an entire file.

```
void main( int argc, char *argv[ ] )
{
   parseEntireFile( "mimefile.txt" );
}
void parseEntireFile( char *szFilename )
{
   nsmail_inputstream_t *pInput;
   if ( file_inputstream_create( szFilename, &pInput ) == MIME_OK )
   {
   mime_message_t *pMessage;
/* Parse the MIME Message */
   if ( parseEntireMessageInputstream( pInput, &pMessage ) == MIME_OK )
   {
      showObject( pMessage, MIME_MESSAGE );
         mime_message_free( pMessage );
   }
   pInput -> close();
   nsStream_free (pInput);
   }
}
```

[Top]

Dynamic Parsing

This section describes the steps involved in using the dynamic parser. Dynamic parsing contrasts with standard MIME parsing, as described in Parsing the Entire Message, in several ways.

- The dynamic parser can parse a message in chunks, rather than in its entirety, in a single operation. The dynamic parser decodes the message on the fly, passing the data to the user right away without waiting for the whole message.
- The dynamic parser returns parsed message data to the caller using callbacks in the data sink. For information about this, see Creating a Data Sink. The `parseEntireMessage` function does not use callbacks; instead, it passes the entire parsed message to the user after parsing is complete.
- The dynamic parser does not decode the Base64/QP-encoded parts of the message. To do this, use the utility functions `mime_decodeBase64` and `mime_decodeQP`.

[Top]

Steps in Dynamic Parsing

Using the dynamic parser involves these operations:

- Create the data sink to specify callback functions for the parser. The MIME data sink contains one call for each piece of information that the parser can return. For example, for a header, it contains a header callback function. See Creating a Data Sink.
- Create a parser object, which takes the data sink as a parameter. See Creating the Dynamic Parser.
- Begin parsing. See Running the Parser.
- As long as there is more data to parse, continue calling the dynamic parsing function that matches the source of the data. Keep calling this function until there is no more data. See Running the Parser.
- When there is no more data to parse, indicate that parsing is complete. These steps are described in Running the Parser.

All dynamic parser functions are defined in `mimeparser.h`.

[Top]

MIME Data Sink Callbacks

Callbacks operate in the same way in MIME as they do in other Messaging Access SDK protocols. However, for SMTP, IMAP4, and POP3, callbacks are tied to server responses to individual functions.

The dynamic parser data sink differs from the response sink in that, with a response sink, the mail application sends a command and gets a server response. In the data sink, when a callback takes place, the data is passed to the sink in callbacks. The callback is dependent upon the data in the input stream.

The MIME data sink contains one callback prototype for each piece of information that the parser can return. The dynamic parser makes callbacks based on the kind of data it finds in its input stream. For example, if the parser finds a header, the result is a header callback. As the parser encounters data, it returns information to the caller through callbacks in the data sink.

For general information about the data sink, response sinks, and callbacks, see SDK Response Sinks for C.

[Top]

Creating a Data Sink

The first step in dynamic parsing is to initialize the MIME data sink, which is defined by the `mimeDataSink_t` structure. The data sink is made up of function pointers and opaque data, which are set to `null` when the sink is initialized. For general information about the data sink, see SDK Response Sinks for C.

After creating the data sink, the application passes it to the parser. As the parser encounters information, it sends this on to the caller through callbacks in the data sink. The MIME data sink contains a call for each piece of information that the parser can return.

To initialize and allocate the data sink, call the `mimeDataSink_new` function and pass in the sink you want to use. If successful, this function returns `NSMAIL_OK`.

```
int mimeDataSink_new ( mimeDataSink_t **pp );
```

You can create parsers with different data sinks, based on what you want the messaging application to do. For example, you can define data sinks that create a brief header, a normal header, or list all header lines, each of which can be invoked with a different parser invocation. To add headers, define the ones your application requires within the data sink structure.

The following section of code creates the data sink and sets the sink function pointers.

```
/* Create the functions for the data sink */

void mimeDataSink_header( mimeDataSinkPtr_t pSink,
                  void *pCallbackObject, char *name, char *value )

{
   sprintf( achTemp, "header()   name = [%s]
                        value = [%s]\n", name, value );
   output( achTemp );
}

void mimeDataSink_contentType( mimeDataSinkPtr_t pSink,
                  void *pCallbackObject, int nContentType )

{
   sprintf( achTemp, "contentType() = [%d]\n", nContentType );
   output( achTemp );

}
   void mimeDataSink_contentSubType( mimeDataSinkPtr_t pSink,
                  void *pCallbackObject, char * contentSubType )

{
   sprintf( achTemp, "contentSubType() = [%s]\n", contentSubType );

output( achTemp );
}

void mimeDataSink_contentTypeParams( mimeDataSinkPtr_t pSink,
                  void *pCallbackObject, char * contentTypeParams
)

{
   sprintf( achTemp, "contentTypeParams() = [%s]\n",
                  contentTypeParams ); output( achTemp );
}

void mimeDataSink_contentID( mimeDataSinkPtr_t pSink,
```

```
                    void *pCallbackObject, char *contentID )
{
   sprintf( achTemp, "contentID() = [%s]\n", contentID );
   output( achTemp );
}

/* Create the data sink */

   mimeDataSink_new (&pDataSink );

/* Add the functions to the data sink */

   pDataSink->header = &mimeDataSink_header;

   pDataSink->contentType = &mimeDataSink_contentType;

   pDataSink->contentSubType = &mimeDataSink_contentSubType;

   pDataSink->contentTypeParams = &mimeDataSink_contentTypeParams;

   pDataSink->contentID = &mimeDataSink_contentID;

/* Continue as required */
```

When a session is finished, you must free any memory associated with the data sink. Use this function:

```
void mimeDataSink_free( mimeDataSink_t **pp );
```

This function frees the data sink structure. The user must free any pointers to opaque data.

After you create the data sink, the next step is Creating the Dynamic Parser.

[Top] [Creating a Data Sink]

Creating the Dynamic Parser

After creating the data sink, the next step is to create a dynamic parser. You can use the `mimeDynamicParser_new` function to create a new parser and identify the data sink to use.

```
int mimeDynamicParser_new( mimeDataSink_t *pDataSink,
                           struct mimeParser **pp);
```

Supply the identifier for the data sink and the parser to use.

The following section of code creates a dynamic parser.

```
/* Initialize sink first, as described in Creating a Data Sink */
if ( mimeDynamicParser_new( pDataSink, &p) != MIME_OK )
```

When a session is finished, you must free any memory associated with the dynamic parser. Use this function:

```
void mimeDynamicParser_free( struct mimeParser **pp );
```

This function frees the dynamic parser structure and its data members.

After you create the dynamic parser, the next step is Running the Parser.

[Top]

Running the Parser

After the dynamic parser object has been created, as described in Creating the Dynamic Parser, parsing can begin. The application instructs the parser to continue the parsing operation until no more data is available, then signals that the parsing is complete.

To begin parsing, use this function:

```
int beginDynamicParse( struct mimeParser *p );
```

This function requires the identifier of the parser created with `mimeDynamicParser_new`.

To continue parsing, use the function that is appropriate for the data source, either an input stream or a data buffer. Continue to call this function until there is no more data left.

Use this function to parse data from an input stream:

```
int dynamicParseInputstream( struct mimeParser *p,
                     struct nsmail_inputstream_t *pInput );
```

This function requires the identifier of the parser and the input stream to use. For the SDK functions that create input and output streams, see Shared Functions.

Use this function to parse data from a data buffer.

```
int dynamicParse( struct mimeParser *p, char *pData, int nLen );
```

This function requires the identifier of the parser in use, the data buffer to use, and the length of the data to parse.

When no more data remains to be parsed, call this function to indicate that parsing is complete:

```
int endDynamicParse( struct mimeParser *p );
```

The parser ends the operation.

To initiate another parsing cycle, you can call `beginDynamicParse` again.

The following section of code creates a dynamic parser, parses data from an input stream, and ends the parse operation.

```
/* Create the dynamic parser; see Creating the Dynamic Parser */

( mimeDynamicParser_new( pDataSink, &p );

   ( file_inputstream_create( szFilename, &pInput );

/* Start dynamic parsing */

      beginDynamicParse( p );

      for ( len = pInput->read( pInput->rock, buffer, BUFFER_SIZE2 );

         len > 0;

         len = pInput->read( pInput->rock, buffer, BUFFER_SIZE2 ) )

/* Continue dynamic parsing until no more data remains */

   {

   if ( dynamicParse( p, buffer, len ) != MIME_OK )

          break;

   }

/* When data is finished, stop the dynamic parser */

endDynamicParse(p);

/* Free the data sink used by the parser */
/* See Creating a Data Sink */

   {

      mimeDataSink_free( &pDataSink );

   }
```

[Top]

MIME Functions by Task

To help you find the information you need more quickly, look for the task you want to perform in the column on the left. Then you can click the function name for further information.

Task	Function for the task
Start dynamic parsing.	beginDynamicParse
Dynamically parse chunks of data from a data buffer.	dynamicParse
Dynamically parse chunks of data from an input stream.	dynamicParseInputstream
Signal the end of dynamic parsing.	endDynamicParse
Return the MIME type information for the specified file, based on file extension.	getFileMIMEType
Delete the message body data.	mime_basicPart_deleteData
Free the basic part including all internal structures.	mime_basicPart_free_all
Return the decoded body-data in a buffer.	mime_basicPart_getDataBuf
Return the decoded body-data in an input stream.	mime_basicPart_getDataStream
Return the size of the body-data in bytes.	mime_basicPart_getSize
Write the byte stream for this part with MIME headers and encoded body-data.	mime_basicPart_putByteStream
Set the body-data for this part from a buffer.	mime_basicPart_setDataBuf
Set the body-data for this part from an input stream.	mime_basicPart_setDataStream
Free a MIME data sink.	mimeDataSink_free

Task	Function for the task
Create a new data sink.	mimeDataSink_new
Decode a Base64-encoded data and write it to an output stream.	mime_decodeBase64
Decode and return a header string.	mime_decodeHeaderString
Decode Quoted Printable-encoded data and write it to an output stream.	mime_decodeQP
Free the dynamic parser.	mimeDynamicParser_free
Create the data sink for the dynamic parser.	mimeDynamicParser_new
Use Base64 encoding to encode message data.	mime_encodeBase64
Encode a header string.	mime_encodeHeaderString
Encode data using QP encoding.	mime_encodeQP
Free a MIME header.	mime_Header_new
Allocate memory for use with the MIME API.	mime_malloc
Free memory allocated by the SDK.	mime_memfree
Add a basic part as the body of the message.	mime_multiPart_addBasicPart
Add a basic part as the body of the message.	mime_message_addBasicPart
Add the message part as the body of the message.	mime_message_addMessagePart
Add the multipart as the body of the message.	mime_message_addMultiPart
Create a message given text-data and a file to attach.	mime_message_create
Delete the body part that makes up the body of this message.	mime_message_deleteBody
Free the message including all internal structures.	mime_message_free_all
Return the body part that makes up the body of the message.	mime_message_getBody

Task	Function for the task
Return the content subtype of this message.	mime_message_getContentSubType
Return the content type of the body of this message.	mime_message_getContentType
Return the content_type parameters of this message.	mime_message_getContentTypeParams
Get the header value, given the header name.	mime_message_getHeader
Encode this part and write it to an output stream.	mime_message_putByteStream
Delete the MIME message that makes up the body of this part.	mime_messagePart_deleteMessage
Free the message-part including all internal structures.	mime_messagePart_free_all
Create a message part from a message structure.	mime_messagePart_fromMessage
Retrieve the message from the message part.	mime_messagePart_getMessage
Write the byte stream for this part with MIME headers and encoded body-data.	mime_messagePart_putByteStream
Set a message as the body of this message-part.	mime_messagePart_setMessage
Add a basic part to a multipart.	mime_multiPart_addBasicPart
Add a file as basic part to the multipart.	mime_multiPart_addFile
Add a message part to the multipart.	mime_multiPart_addMessagePart
Add a message part to the multipart.	mime_multiPart_addMultiPart
Delete the body part at the specified index from this multipart.	mime_multiPart_deletePart
Free the multipart including all internal structures.	mime_multiPart_free_all
Return the body part at the specified index.	mime_multiPart_getPart

Task	Function for the task
Return the count of the body parts in this multipart.	mime_multiPart_getPartCount
Write the byte stream for this part with MIME headers and encoded body-data.	mime_multiPart_putByteStream
Parse an entire message from a data buffer.	parseEntireMessage
Parse an entire message from an input stream.	parseEntireMessageInputstream

[Top] [MIME Functions by Task]

Chapter

4

Receiving Mail with IMAP4

This chapter is an overview of using IMAP4 (Internet Message Access Protocol 4) to retrieve and manage messages remotely.

- The IMAP4 Protocol
- IMAP4 Function Callback Mapping
- Creating a Response Sink
- Creating a Client
- Connecting to a Server
- Logging In and Out
- Checking for New Messages
- Searching for Messages
- Fetching Message Data
- Closing a Mailbox
- IMAP4 Functions by Task

[Top]

The IMAP4 Protocol

IMAP4 (Internet Message Access Protocol, Version 4), which was developed at the University of Washington, allows clients to retrieve and manage their email messages remotely. This can be very helpful to users who access mail on several different computers.

IMAP4 also provides these capabilities:

- Filing: You can create folders, called “mailboxes,” on the server, manage mail messages on the server (search, delete, rename), and transfer messages from one folder to another on the server
- Searching: You can find the messages that meet specified criteria on the server without downloading messages to the client

To send mail, use SMTP (Simple Mail Transport Protocol). For more information, see Chapter 2, “Sending Mail with SMTP.”

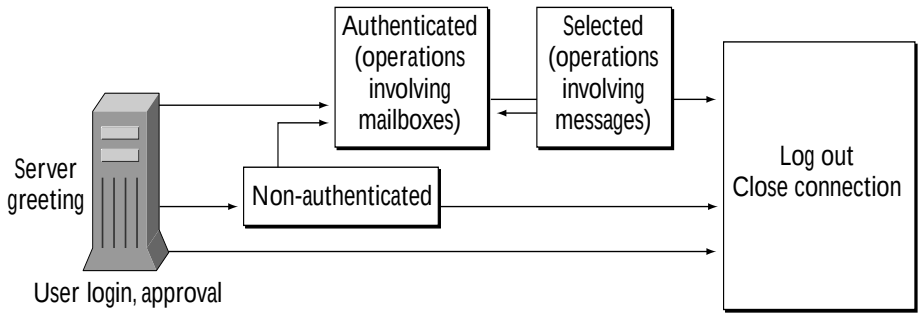
For detailed information about IMAP4, consult one of the RFCs listed, with links, in IMAP4 RFCs.

[Top]

IMAP4 Session States

An IMAP4 session progresses through several stages, or states. Within each state, only certain commands are possible.

Figure 4.1 IMAP4 Session States



Note: All states can result in logging out and closing the connection in response to the logout command, server shutdown, or a closed connection.

Table 4.1 IMAP4 Session States and Commands

Session State	Commands
All States	Commands: CAPABILITY, LOGOUT, NOOP
Non-Authenticated	Before login. User login, approval. Command: LOGIN
Authenticated	User is logged in, can perform operations involving mailboxes and mailbox management. Commands: APPEND, CREATE, DELETE, EXAMINE, LIST, LSUB, RENAME, SELECT, STATUS, SUBSCRIBE, UNSUBSCRIBE
Selected	Operations involving messages. Commands: CHECK, CLOSE, COPY, EXPUNGE, FETCH, SEARCH, STORE, UID

The client must keep track of the current session state in order to know which commands are valid.

For a table of SDK-supported IMAP4 protocol commands that lists the state in which each can be called, see Supported IMAP4 Internet Protocol Commands. For detailed information about IMAP4 and IMAP4 session states, consult one of the RFCs listed, with links, in IMAP4 RFCs.

[Top]

Steps in an IMAP4 Session

Generally, a messaging application follows these steps when using IMAP4 to receive mail and manage mailboxes and messages. These steps are listed below with links to more detailed descriptions.

Step	Section with details
Create a response sink.	Creating a Response Sink
Create a client.	Creating a Client
Connect to the server.	Connecting to a Server
Log in.	Logging In and Out
Check for new messages.	Checking for New Messages
Select a mailbox.	Searching for Messages
Fetch a new message.	Fetching Message Data
Perform other message and mailbox management tasks.	IMAP4 Functions by Task
Close the mailbox.	Closing a Mailbox

[Top]

IMAP4 Function Callback Mapping

Callbacks are associated with many IMAP4 functions. For general information about the response sink and callbacks, see SDK Response Sinks for C. These callbacks are invoked by the client's `imap4_processResponses` function.

Functions with multi-line responses map to two or more callbacks. For example, when a function is mapped to three callbacks, the first provides a notification of the start of the operation, the second of the response, and the third that the operation is complete.

Many IMAP4 functions generate a tag (`out_ppTagID`) that you can use to help match the command and the response associated with it within the `imap4Sink_t.taggedLine` response. Remember to use `imap4Tag_free` to free the memory associated with these tags.

If a server error occurs, the error callback is invoked.

Table 4.2 shows which IMAP4 functions are mapped to callbacks in the `imap4Sink_t` structure. Table 4.3 shows which functions do not map to callbacks.

Table 4.2 Functions with Callbacks

Functions	Possible Responses, Mapped to Callbacks
General Commands	
`imap4_connect`	`error`, `ok`
`imap4_disconnect`	`bye`
`imap4_sendCommand`	`rawResponse`, `taggedLine`, `error`
Non-Authenticated State Commands	
`imap4_capability`	`capability`, `taggedLine`, `error`
`imap4_noop`	`exists`, `expunge`, `recent`, `fetchStart`, `fetchFlags`, `fetchEnd`, `taggedLine`, `error`
`imap4_login`	`taggedLine`, `error`
`imap4_logout`	`bye`, `taggedLine`, `error`
Authenticated State Commands	
`imap4_append`	`taggedLine`, `error`
`imap4_create`	`taggedLine`, `error`

Functions	Possible Responses, Mapped to Callbacks
imap4_delete	taggedLine, error
imap4_examine	flags, exists, recent, ok, taggedLine, error
imap4_list	list, taggedLine, error
imap4_lsub	lsub, taggedLine, error
imap4_rename	taggedLine, error
imap4_select	flags, exists, recent, ok, taggedLine, error
imap4_status	statusMessages, statusRecent, statusUidnext, statusUidvalidity, statusUnseen, taggedLine, error
imap4_subscribe	taggedLine, error
imap4_unsubscribe	taggedLine, error
Selected State Commands	
imap4_check	taggedLine, error
imap4_close	taggedLine, error
imap4_copy	taggedLine, error
imap4_uidCopy	taggedLine, error
imap4_expunge	expunge, taggedLine, error
imap4_fetch	fetchStart, fetchEnd, fetchSize, fetchData, fetchFlags, fetchBodyStructure, fetchEnvelope, fetchInternalDate, fetchHeader, fetchUid, taggedLine, error
imap4_uidFetch	fetchStart, fetchEnd, fetchSize, fetchData, fetchFlags, fetchBodyStructure, fetchEnvelope, fetchInternalDate, fetchHeader, fetchUid, taggedLine, error
imap4_search	searchStart, search, searchEnd, taggedLine, error
imap4_uidSearch	searchStart, search, searchEnd, taggedLine, error
imap4_store	taggedLine, error
imap4_uidStore	taggedLine, error
Extended IMAP Commands	

Functions	Possible Responses, Mapped to Callbacks
imap4_namespace	nameSpaceStart, nameSpacePersonal, nameSpaceOtherUsers, nameSpaceShared, nameSpaceEnd, taggedLine, error
imap4_setACL	taggedLine, error
imap4_deleteACL	taggedLine, error
imap4_getACL	aclStart, aclIdentifierRight, aclEnd, taggedLine, error
imap4_myRights	myRights, taggedLine, error
imap4_listRights	listRightsStart, listRightsRequiredRights, listRightsOptionalRights, listRightsEnd, taggedLine, error

Table 4.3 Functions without Callbacks

Functions Without Callbacks	
imap4_free	imap4_setResponseSink
imap4_get_option	imap4_setTimeout
imap4_initialize	imap4Sink_free
imap4_processResponses	imap4Sink_initialize
imap4_setChunkSize	imap4Tag_free
imap4_set_option	

[Top] [IMAP4 Function Callback Mapping]

Creating a Response Sink

The first step in starting an IMAP4 session is to initialize the IMAP4 response sink, which is defined by the imap4Sink_t structure. The response sink is made up of function pointers and opaque data. Initializing the sink sets its function pointers and opaque data to null. For general information about the response sink, see SDK Response Sinks for C.

To initialize and allocate the response sink, call the imap4Sink_initialize function. If successful, this function returns NSMAIL_OK. Use this syntax:

```
int imap4Sink_initialize(imap4Sink_t** out_ppimap4Sink);
```

The following section of code initializes the response sink and sets the sink function pointers.

```
int l_retCode;

imap4Sink_t* l_pimap4Sink = NULL;

l_retCode = imap4Sink_initialize( &l_pimap4Sink);

if(l_retCode != NSMAIL_OK)   { /*Deal with error */ }

setSinkFunctions(l_pimap4Sink); /* Set sink function pointers */
```

After you create the response sink, the next step is Creating a Client.

When a session is finished, you must free any memory associated with the response sink. To free the response sink and its data members, use imap4Sink_free:

```
int imap4Sink_free(imap4Sink_t** out_ppimap4Sink);
```

When this function returns, the response sink is set to null. The user must free any opaque data.

[Top] [Creating a Response Sink]

Creating a Client

The IMAP4 client uses the `imap4Client_t` structure to communicate with an IMAP4 server. To initialize and allocate the `imap4Client_t` structure and set the response sink for the client's use, call the `imap4_initialize` function.

When you create the client structure, you pass in the identifier. Use this syntax:

```
int imap4_initialize(imap4Client_t ** in_ppimap4,
                     imap4Sink_t* in_pimap4Sink);
```

The following section of code creates a client.

```
/* Initialize sink first as described in Creating a Response Sink */

imap4Client_t * l_pimap4Client = NULL;

l_retCode = imap4_initialize(&l_pimap4Client, l_pimap4Sink);

if(l_retCode != NSMAIL_OK)   { /*Deal with error*/ }
```

After you initialize the client, the next step is Connecting to a Server.

When a session is finished, you must free any memory associated with the client. To free the client structure and its data members, use this function:

```
int imap4_free(imap4Client_t ** in_ppimap4);
```

The `in_ppimap4` parameter represents the IMAP4 client. When this function returns, the client structure is set to `null`.

[Top]

Connecting to a Server

Before retrieving mail, the client must connect with the server through a service port. To connect to the server, use the `imap4_connect` function:

```
int imap4_connect(imap4Client_t * in_pimap4,
                  const char * in_host,
                  unsigned short in_port,
                  char** out_ppTagID);
```

Supply the identifier of the IMAP4 client that wants to connect with the server, the identifier of the host server, and the connection port to use.

This function generates a tag that you can use to help match the command and the response associated with it. Remember to use imap4Tag_free to free the memory associated with the tag.

The following section of code connects the client to the server.

```
/* After Creating a Response Sink and Creating a Client */

char * l_szServer = “Server Name”;

char * l_szTagID = “NULL”;

l_retCode = imap4_connect(l_pimap4Client, l_szServer, 143, &l_szTagID);

if(l_retCode != NSMAIL_OK) { /*Deal with error*/ }

l_retCode = imap4_processResponses(l_pimap4Client);

if(l_retCode != NSMAIL_OK) { /*Deal with error*/ }

imap4Tag_free(&l_szTagID);
```

During the connect process, you can find out what extensions the server supports. For more information, see Determining Server Capabilities. You also might want to log in. For more information, see Logging In and Out.

When a session is finished, free any memory associated with the client. To disconnect the client from the server and close the socket connection, use this function:

```
int imap4_disconnect( imap4Client_t * in_pimap4 );
```

The in_pimap4 parameter represents the IMAP4 client. You could use this function as part of a Cancel operation while retrieving a message. Remember that you do not call imap4_processResponses after imap4_disconnect.

Note For the callback mapping for these functions, see IMAP4 Function Callback Mapping. §

[Top]

Determining Server Capabilities

To retrieve a listing of the capabilities that are supported by the server, call the `imap4_capability` function:

```
int imap4_capability(imap4Client_t* in_pimap4,
                     char** out_ppTagID);
```

This function calls the CAPABILITY IMAP4 protocol command, which can be issued in any session state, but is usually issued after connecting to the server.

This function generates a tag that you can use to help match the command and the response associated with it. Remember to use `imap4Tag_free` to free the memory associated with the tag.

Note For this function's callback mapping, see IMAP4 Function Callback Mapping. §

The following section of code retrieves a list of server capabilities.

```
/* After Connecting to a Server */

l_retCode = imap4_capability(l_pimap4Client, &out_pTagID);

if(l_retCode != NSMAIL_OK)   { /*Deal with error*/ }

l_retCode = imap4_processResponses(l_pimap4Client);

if(l_retCode != NSMAIL_OK)   { /*Deal with error*/ }

imap4Tag_free(&out_pTagID);
```

[Top]

Logging In and Out

Once the client is connected to the server, the user can log in. Login identifies the client to the server. Logging in requires the identifier of the IMAP4 client, as well as the user's ID and plain text password. Use this function:

```
int imap4_login(imap4Client_t* in_pimap4,
                const char* in_user,
                const char* in_password,
                char** out_ppTagID);
```

The function sends the LOGIN IMAP4 protocol command, which can be issued during the Non-Authenticated state. Successful login moves the IMAP4 session to the Authenticated state, where the user can search for messages and manage messages on the server.

This function generates a tag that you can use to help match the command and the response associated with it. Remember to use `imap4Tag_free` to free the memory associated with the tag.

The following section of code logs the user in with user name and password.

```
l_retCode = imap4_login(l_pimap4Client, l_szUser, l_szPassword, NULL);
if(l_retCode != NSMAIL_OK)   { /*Deal with error*/ }
l_retCode = imap4_processResponses(l_pimap4Client);
if(l_retCode != NSMAIL_OK)   { /*Deal with error*/ }
imap4Tag_free(&l_szTagID);
```

After you log in, the next step is Checking for New Messages.

To log out at the end of a session, use this function:

```
int imap4_logout(imap4Client_t* in_pimap4, char** out_ppTagID);
```

Remember to use `imap4Tag_free` to free the memory associated with the tag this function generates.

The following section of code logs the user out.

```
l_retCode = imap4_logout(l_pimap4Client, &l_szTagID);
if(l_retCode != NSMAIL_OK)   { /*Deal with error*/ }
l_retCode = imap4_processResponses(l_pimap4Client);
if(l_retCode != NSMAIL_OK)   { /*Deal with error*/ }
imap4Tag_free(&l_szTagID);
```

Note For the callback mapping for these functions, see IMAP4 Function Callback Mapping. §

[Top]

Checking for New Messages

Most IMAP4 servers check for messages whenever a command is issued. In the absence of commands, the server does not check for messages and may disconnect. To keep the server open indefinitely and check for messages periodically, the developer can call imap4_noop at set intervals.

The imap4_noop function is ideal for polling for new mail and ensuring that the server connection is still active. imap4_noop does nothing in itself, so it only produces the side effect of resetting the autologout timer inside the server and retrieving unsolicited server responses, which all commands do. The server responses may indicate the arrival of new messages or a change in the attributes of an existing message. Use this function:

```
int imap4_noop(imap4Client_t* in_pimap4, char** out_ppTagID);
```

The in_pimap4 parameter represents the IMAP4 client. The function generates a tag to associate with the command. Remember to use imap4Tag_free to free the memory associated with the tag.

Note For this function's callback mapping, see IMAP4 Function Callback Mapping. §

The following section of code uses imap4_noop to check for messages.

```
l_retCode = imap4_noop(l_pimap4Client, &out_pTagID);
if(l_retCode != NSMAIL_OK)   { /*Deal with error*/ }
l_retCode = imap4_processResponses(l_pimap4Client);
if(l_retCode != NSMAIL_OK)   { /*Deal with error*/ }
imap4Tag_free(&out_pTagID);
```

After checking for new messages, the next step is Searching for Messages.

[Top]

Searching for Messages

IMAP4 provides two ways to search for messages while in the Selected state. Two functions, `imap4_search` and `imap4_uidsearch`, search the currently selected mailbox and return the message numbers of messages that match a search key. These numbers can be used in turn to fetch the messages themselves.

You can supply one or more of the search keys defined in RFC 2060. Place more than one search key in a parenthesized list. For a table that summarizes the data items you can use to search, see Search Function Search Keys.

The `imap4_search` function searches the mailbox for messages that match the search criteria and returns their message numbers:

```
int imap4_search( imap4Client_t* in_pimap4,
                  const char* in_criteria,
                  char** out_ppTagID);
```

This function sends the SEARCH IMAP4 protocol command, which can be issued in the Selected session state.

The `imap4_uidsearch` function searches the mailbox for messages specified with a unique identifier:

```
int imap4_uidSearch( imap4Client_t* in_pimap4,
                     const char* in_criteria,
                     char** out_ppTagID);
```

This function uses the UID IMAP4 protocol command to specify that the SEARCH command uses unique message identifiers rather than sequence numbers.

This function generates a tag that you can use to help match the command and the response associated with it. Remember to use `imap4Tag_free` to free the memory associated with the tag.

Note For the callback mapping for these functions, see IMAP4 Function Callback Mapping. §

The following section of code searches for messages that have the SUBJECT “afternoon meeting.”

```
int l_retCode = 0;

l_retCode = imap4_search(l_pimap4Client, "SUBJECT \"afternoon
meeting\"",
```

```
&out_pTagID);
if(l_retCode != NSMAIL_OK)   { /*Deal with error*/ }
l_retCode = imap4_processResponses(l_pimap4Client);
if(l_retCode != NSMAIL_OK)   { /*Deal with error*/ }
imap4Tag_free(&out_pTagID);
```

After locating messages, the next step is Fetching Message Data.

[Top]

Fetching Message Data

IMAP4 provides two functions that fetch messages while in the Selected state, `imap4_fetch` and `imap4_uidfetch` both search the currently selected mailbox and retrieve the data specified by the fetch criteria.

When you fetch messages, you supply the message set (mailbox) and one or more fetch criteria, placing more than one data item in a parenthesized list. The fetch criteria determine the information that is returned. You can fetch one or more of the data items defined in in RFC 2060. For a table that summarizes these data items, see RFC 2060 Fetch Data Items.

The `imap4_fetch` and `imap4_uidfetch` functions retrieve the data specified by the fetch criteria for the given message set.

This function performs a fetch:

```
int imap4_fetch(imap4Client_t* in_pimap4,
                const char* in_msgSet,
                const char* in_fetchCriteria,
                char** out_ppTagID);
```

The `imap4_uidfetch` function performs a fetch using unique identifiers for messages:

```
int imap4_uidFetch(imap4Client_t* in_pimap4,
                   const char* in_msgSet,
                   const char* in_fetchCriteria,
                   char** out_ppTagID);
```

This function uses the UID IMAP4 protocol command to specify that the FETCH command uses unique message identifiers rather than sequence numbers.

Both functions generate a tag that you can use to help match the commands and the responses associated with them. Remember to use `imap4Tag_free` to free the memory associated with the tags.

Note For the callback mapping for these functions, see IMAP4 Function Callback Mapping. §

The following section of code fetches the body of the message specified by the message number.

```
l_retCode = imap4_fetch(l_pimap4Client, "1:2", "BODY[]", &out_pTagID);
if(l_retCode != NSMAIL_OK)   { /*Deal with error*/ }
l_retCode = imap4_processResponses(l_pimap4Client);
if(l_retCode != NSMAIL_OK)   { /*Deal with error*/ }
imap4Tag_free(&out_pTagID);
```

[Top]

Closing a Mailbox

To close a mailbox, use the `imap4_close` function. This function sends the CLOSE IMAP4 protocol command, which closes the mailbox and removes any messages marked with the `\Deleted` flag. You can close the mailbox without logging out. In this case, the session moves to the parent mailbox. Use this syntax:

```
int imap4_close(imap4Client_t* in_pimap4, char** out_ppTagID);
```

If you need to permanently delete messages without closing, call the `imap4_expunge` function.

This function generates a tag that you can use to help match the command and the response associated with it. Remember to use `imap4Tag_free` to free the memory associated with the tag.

Note For this function's callback mapping, see IMAP4 Function Callback Mapping. §

The following section of code closes a mailbox.

```
l_retCode = imap4_close(l_pimap4Client, &out_pTagID);
if(l_retCode != NSMAIL_OK)   { /*Deal with error*/ }
l_retCode = imap4_processResponses(l_pimap4Client);
if(l_retCode != NSMAIL_OK)   { /*Deal with error*/ }
imap4Tag_free(&out_pTagID);
```

[Top]

IMAP4 Functions by Task

To help you find the information you need more quickly, look for the task you want to perform in the column on the left. Then you can click the function name for further information.

Note All functions that allocate tags must use `imap4Tag_free` to free the associated memory. §

Task	Function for the task
General Functions	
Connect to the IMAP server `in_host` through the port `in_portNumber`.	`imap4_connect`
Disconnect from the IMAP4 server.	`imap4_disconnect`
Free the IMAP4 client structure and its data members.	`imap4_free`
Get the IO model.	`imap4_get_option`
Allocate space for the IMAP4 client structure.	`imap4_initialize`
Process the server responses for all API commands executed prior to this command.	`imap4_processResponses`
Send the specified command to the IMAP4 server.	`imap4_sendCommand`
Set the size of the message data chunk returned in `fetchData`.	`imap4_setChunkSize`

Task	Function for the task
Set the IO model.	imap4_set_option
Set the response sink to the specified sink.	imap4_setResponseSink
Set the amount of time allowed to wait for data within imap4_processsResponses before returning control to the user.	imap4_setTimeout
Free the response sink.	imap4Sink_free
Allocate space for the IMAP4 sink structure.	imap4Sink_initialize
Free tags generated by a function.	imap4Tag_free
Non-Authenticated State Functions	
Request a listing of capabilities that the server supports.	imap4_capability
Identify the client to the server.	imap4_login
Tell the server that the client is done with the connection.	imap4_logout
Issue a command that always succeeds and does nothing.	imap4_noop
Authenticated State Functions	
Append a message to the specified mailbox.	imap4_append
Create a mailbox with the given name.	imap4_create
PERMANENTLY remove the mailbox with the given name.	imap4_delete
Select a server mailbox for read-only access.	imap4_examine
List the mailboxes.	imap4_list
Get a subset of user-defined “active” or “subscribed” names.	imap4_lsub
Rename a mailbox.	imap4_rename
Select a mailbox on the server.	imap4_select
Request the status of a particular mailbox.	imap4_status
Add a mailbox name to the server's set of “active” or “subscribed” mailboxes.	imap4_subscribe

Task	Function for the task
Remove a mailbox from server's subscribed mailbox list.	`imap4_unsubscribe`
Selected State Functions	
Request a checkpoint of the currently selected mailbox.	`imap4_check`
Close the mailbox.	`imap4_close`
Copy the specified message(s) from the currently selected mailbox to the end of the specified destination mailbox.	`imap4_copy`
Remove all messages flagged for deletion.	`imap4_expunge`
Retrieve data specified by the fetch criteria for the given message set and return the corresponding message sequence numbers.	`imap4_fetch`
Search the mailbox for messages that match your search criteria and retrieve the corresponding message sequence numbers.	`imap4_search`
Alter data associated with a message in the mailbox.	`imap4_store`
Copy messages specified with a unique identifier.	`imap4_uidCopy`
Fetch messages specified with a unique identifier and retrieve the corresponding unique identifiers.	`imap4_uidFetch`
Search the mailbox for messages that match your search criteria and retrieve the corresponding unique identifiers.	`imap4_uidSearch`
Update message data associated with a unique identifier.	`imap4_uidStore`
Extended IMAP4 Functions	
Remove an identifier from the access control list for a mailbox.	`imap4_deleteACL`
Get the access control list for a mailbox.	`imap4_getACL`
Find which rights you can grant a specified user to a particular mailbox.	`imap4_listRights`
Find which rights a user has to a mailbox.	`imap4_myRights`

Task	Function for the task
Find the prefixes of namespaces used by a server for personal mailboxes, other user's mailboxes, and shared objects.	`imap4_namespace`
Change the access control list to grant permissions to an identifier for a mailbox.	`imap4_setACL`

[Top] [IMAP4 Functions by Task]

Chapter

5

Receiving Mail with POP3

This chapter is an overview of using POP3 (Post Office Protocol 3) to download messages to a client.

- The POP3 Protocol
- POP3 Function Callback Mapping
- Creating a Response Sink
- Creating a Client
- Connecting to a Server
- Logging In
- Getting Message Count
- Listing Messages
- Retrieving Message Headers
- Retrieving a Message
- Ending the Session

[Top]

The POP3 Protocol

POP3 (Post Office Protocol 3) retrieves mail from mailboxes on a remote server. The server retains messages until the client requests them.

Unlike IMAP4, POP3 only receives mail. IMAP4 provides the capabilities of POP3 along with the ability to move messages back and forth between client and server, and manage mailboxes on the server. For information about IMAP4, see Chapter 4, “Receiving Mail with IMAP4.”

POP3 commands use the ASCII character set. They are made up of a keyword, followed by any parameters the command has, and ending with “<CRLF>.<CRLF>.” Commands are line-oriented and can return a single or multi-line response.

For detailed information about POP3, see RFC 1939: “Post Office Protocol - Version 3.”

[Top]

POP3 Session States

A POP3 session progresses through three stages, or states. Within each state, only certain commands are possible.

Figure 5.1 POP3 Session States

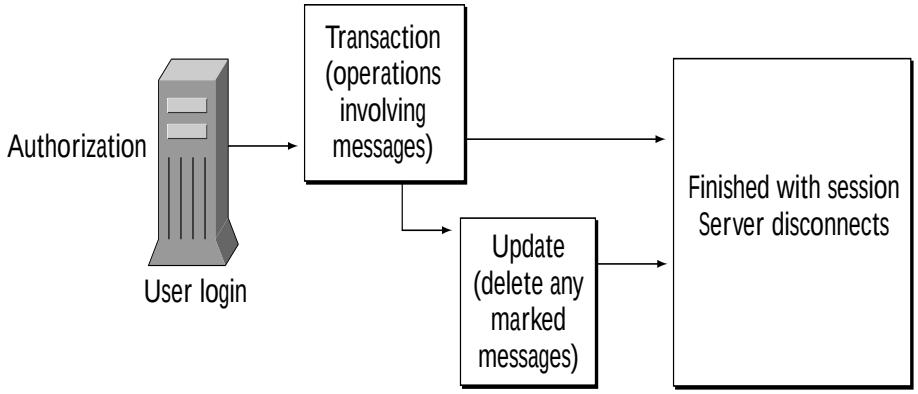


Table 5.1 POP3 Session States and Commands

Session State	Commands
Authorization	User login, password approval, quit. Commands: USER, PASS, QUIT
Transaction	Operations involving messages. Commands: STAT, LIST, RETR, DELE, NOOP, RSET, TOP, UIDL
Update	Delete any marked messages. The session enters this state after a QUIT command, deletes messages marked for deletion, then quits. Commands: QUIT

For a table of SDK-supported POP3 protocol commands that lists the state in which each can be called, see Supported POP3 Internet Protocol Commands. For detailed information about POP3 and POP3 session states, see POP3 RFCs.

[Top]

Steps in a POP3 Session

Generally, a messaging application follows these steps when using POP3 to receive mail. These steps are listed below with links to more detailed descriptions.

Step	Section with details
Initialize the response sink.	Creating a Response Sink
Create a client.	Creating a Client
Connect to the server.	Connecting to a Server
Log in to the server.	Logging In
Get the message count.	Getting Message Count
List messages on the server.	Listing Messages
Retrieve the message headers.	Retrieving Message Headers
Retrieve messages themselves.	Retrieving a Message
End the POP3 session.	Ending the Session

[Top]

POP3 Response Codes

When a client sends a POP3 command, a text response code is returned with descriptive text. The response can either be single- or multi-line.

Table 5.2 POP3 Response Types

Response Type	Response Code and Description
Single line	• +OK (Success), followed by descriptive text, for example, "+OK message deleted" for the delete operation. Responses are mapped to appropriate callbacks. • -ERR (failure), followed by descriptive text, for example, "-ERR no such message" for the delete operation. Mapped to error callback.
Multi-line	**First line:** Like single-line response: • +OK (Success), followed by descriptive text, for example, "+OK message follows" for the retrieve operation. Responses are mapped to appropriate callbacks. • -ERR (failure), followed by descriptive text, for example, "-ERR no such message" for the retrieve operation. Mapped to error callback. **Subsequent lines:** More information about the condition. **Final line**: . (dot) and <CRLF>. (Not considered part of the response.) **Note:** If an error occurs on a multi-line response, a single line is returned.

For a list of functions and their associated callbacks, see POP3 Function Callback Mapping.

[Top]

POP3 Function Callback Mapping

Callbacks are associated with many POP3 functions. For general information about the response sink and callbacks, see SDK Response Sinks for C.

The pop3Sink_t structure contains callbacks for each client call. The client's processResponses function invokes the interface function that corresponds to the client call. Functions with multi-line responses map to more than one

callback. The first type of callback indicates the start of a notification. The second type passes back multi-line data. The third type of callback provides a notification that the operation is complete.

If a server error occurs, the error callback is invoked.

Table 5.3 shows which POP3 functions are mapped to callbacks in the pop3Sink_t structure. Table 5.4 shows functions that do not map to callbacks.

Table 5.3 Functions with Callbacks

Functions	Possible Responses, Mapped to Callbacks
pop3_connect	connect, error
pop3_delete	dele, error
pop3_list	listStart, list, listComplete, error
pop3_listA	listStart, list, listComplete, error
pop3_noop	noop, error
pop3_pass	pass, error
pop3_quit	quit, error
pop3_reset	reset, error
pop3_retrieve	retrieveStart, retrieve, retrieveComplete, error
pop3_sendCommand	sendCommandStart, sendCommand, sendCommandComplete, error
pop3_stat	stat, error
pop3_top	topStart, top, topComplete, error
pop3_uidList	uidListStart, uidList, uidListComplete, error
pop3_uidListA	uidListStart, uidList, uidListComplete, error
pop3_user	user, error
pop3_xAuthList	xAuthListStart, xAuthList, xAuthListComplete, error
pop3_xAuthListA	xAuthListStart, xAuthList, xAuthListComplete, error
pop3_xSender	xSender, error

Table 5.4 Functions without Callbacks

Functions Without Callbacks	
pop3_free	pop3_setResponseSink
pop3_disconnect	pop3_setTimeout
pop3_get_option	pop3_set_option
pop3_initialize	pop3Sink_free
pop3_processResponses	pop3Sink_initialize
pop3_setChunkSize	

[Top] [POP3 Function Callback Mapping]

Creating a Response Sink

The first step in starting a POP3 session is to initialize the POP3 response sink, which is defined by the pop3Sink_t structure. The response sink is made up of function pointers and opaque data. Initializing the sink sets its function pointers and opaque data to null. For general information about the response sink, see SDK Response Sinks for C.

To initialize and allocate the response sink, call the pop3Sink_initialize function and supply the sink you want the POP3 client to use. If successful, this function returns NSMAIL_OK. Use this syntax:

```
int pop3Sink_initialize( pop3Sink_t ** out_ppPOP3Sink )
```

The following section of code initializes the response sink and sets the sink function pointer.

```
int l_retCode = 0;

pop3Sink_t * l_pPOP3Sink = NULL;

l_retCode = pop3Sink_initialize(&l_pPOPSink);

if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }

setSinkFunctions(l_pPOP3Sink);
```

When a session is finished, you must free any memory associated with the response sink. To free the response sink and its data members, use this function:

```
void pop3Sink_free( pop3Client_t ** in_ppPOP3Sink );
```

When this function returns, the response sink is set to `null`. The user must free any opaque data.

After you create the response sink, the next step is Creating a Client.

[Top] [Creating a Response Sink]

Creating a Client

The POP3 client uses the `pop3Client_t` structure to communicate with a POP3 server. To initialize and allocate the `pop3Client_t` structure and set the response sink for the client's use, call the `pop3_initialize` function.

When you create the client structure, you supply the identifier of the POP3 client you are creating, along with an initialized response sink, as described in Creating a Response Sink. Use this syntax:

```
int pop3_initialize( pop3Client_t ** out_ppPOP3,
                     pop3SinkPtr_t in_pPOP3Sink );
```

The following section of code creates a client.

```
/* Initialize sink first as described in Creating a Response Sink */

pop3Client_t * l_pPOP3Client = NULL;

l_retCode = pop3_initialize(&l_pPOP3Client, l_pPOP3Sink);

if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
```

When a session is finished, you must free any memory associated with the client. To free the client structure and its data members, use this function:

```
void pop3_free( pop3Client_t ** in_pPOP3 );
```

The `in_pPOP3` parameter represents the POP3 client. When this function returns, the client structure is set to `null`.

After you initialize the client, the next step is Connecting to a Server.

[Top]

Connecting to a Server

Before it can retrieve mail, the client must connect with the server through a service port. To connect to the server, use the `pop3_connect` function.

Supply the identifier of the POP3 client that wants to connect with the server. If the server is using the default port for the POP3 protocol (port 110), you can enter 0 as the value of the port parameter. The SDK function performs some error-checking. Use this syntax:

```
int pop3_connect( pop3Client_t * in_pPOP3,
                  const char * in_server,
                  unsigned short  in_port );
```

Note For this function's callback mapping, see POP3 Function Callback Mapping. §

The following section of code connects the client to the server.

```
/* After Creating a Response Sink and Creating a Client */

int l_retCode = 0;

char* l_szServer = "pop3Server.com";

pop3Client_t * l_pPOP3Client = NULL;

l_retCode = pop3_connect(l_pPOP3Client, l_szServer, 110);

if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }

l_retCode = pop3_processResponses(l_pPOP3Client);

if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
```

After connecting, the next step is Logging In.

When a session is finished, you must free any memory associated with the client. To disconnect the client from the server and close the socket connection, use this function:

```
int pop3_disconnect( pop3Client_t * in_pPOP3 );
```

The `in_pPOP3` parameter represents the POP3 client. This function could be useful as part of a Cancel operation while retrieving a message. Remember that you do not call `pop3_processResponses` after `pop3_disconnect`.

Note For the callback mapping for these functions, see POP3 Function Callback Mapping. §

[Top]

Logging In

Once the client is connected to the server, the user can log in. Login identifies the client to the server. Logging in requires the identifier of the POP3 client, as well as the user's ID and plain text password.

First, call `pop3_user` and pass the POP3 client and the user name. Use this function:

```
int pop3_user( pop3Client_t * in_pPOP3, const char * in_user );
```

This function sends the USER POP3 protocol command.

To submit the user password, call the `pop3_pass` function:

```
int pop3_pass( pop3Client_t * in_pPOP3,
            const char * in_password );
```

This function sends the PASS POP3 protocol command.

While these commands execute, the session is in Authorization state. Successful completion moves the session to the Transaction state. For a list of states and commands that can be executed in each state, see POP3 Session States.

A successful login moves the POP3 session from the Authorization state to the Transaction state, where the user can perform a number of operations.

Note For the callback mapping for these functions, see POP3 Function Callback Mapping. §

The following section of code logs the user in with user name and password.

```
/* User id sequence */
```

```
int l_retCode = 0;
char* l_szUser = "test1";
char* l_szPassword = "test1Password";
pop3Client_t * l_pPOP3Client = NULL;
l_retCode = pop3_user(l_pPOP3Client, l_szUser);
if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
l_retCode = pop3_processResponses(l_pPOP3Client);
if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
/* Password sequence */
l_retCode = pop3_pass(l_pPOP3Client, l_szPassword);
if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
l_retCode = pop3_processResponses(l_pPOP3Client);
if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
```

After you log in, you can perform any Transaction state operations. The next step is Retrieving a Message.

[Top]

Getting Message Count

In the POP3 Transaction state, the client can find out how many messages are present by requesting the status of the mail drop or mailbox. The `pop3_stat` function gets the mailbox status for a given client by issuing the STAT protocol command. Use this function:

```
int pop3_stat( pop3Client_t * in_pPOP3 );
```

The status returned includes the number of messages present and the octet size of the mail drop.

Note For this function's callback mapping, see POP3 Function Callback Mapping. §

The following section of code counts the messages in the client's mailbox.

```
int l_retCode = 0;
```

```
pop3Client_t * l_pPOP3Client = NULL;
l_retCode = pop3_stat(l_pPOP3Client);
if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
l_retCode = pop3_processResponses(l_pPOP3Client);
if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
```

[Top]

Listing Messages

To list messages while in the POP3 Transaction state, call either of two functions, depending on whether you want to list all of the messages in a mailbox or a specific message.

To go through all the messages in the mailbox and generate a list of messages, use the `pop3_list` function:

```
int pop3_list( pop3Client_t * in_pPOP3 );
```

This function takes only the POP3 client as a parameter. It sends the LIST POP3 protocol command.

To list only the message specified by the message number, use the `pop3_listA` function:

```
int pop3_listA( pop3Client_t * in_pPOP3, int in_messageNumber );
```

Supply the POP3 client and a message number as parameters. This function sends the LIST [arg] POP3 protocol command.

Note For the callback mapping for these functions, see POP3 Function Callback Mapping. §

The following section of code retrieves the email address of the sender along with authenticated messages.

```
int l_retCode = 0;
pop3Client_t * l_pPOP3Client = NULL;
l_retCode = pop3_list(l_pPOP3Client);
```

```
if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }

l_retCode = pop3_processResponses(l_pPOP3Client);

if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
```

[Top]

Retrieving Message Headers

In the Transaction state, the user can preview mailbox contents or part of a long message before deciding to download it by listing the headers plus some of the lines of the body. The user (or the mail application) determines the number of message lines to retrieve.

To identify the message, supply the number of the message and the number of body lines to retrieve. Use the `pop3_top` function, which issues the TOP protocol command:

```
int pop3_top(pop3Client_t * in_pPOP3,
             int in_messageNumber, int in_lines );
```

To retrieve all headers for a given mailbox, combine `pop3_top` with a call to `pop3_stat`. First, call `pop3_stat` to find the number of messages in the mailbox. When you get the number, for example, 10, call `pop3_top` once, passing each message number in turn, until you get to the total (in this case, 10). For the `in_lines` parameter, use a value of 0 so that no body lines are returned.

Note For this function's callback mapping, see POP3 Function Callback Mapping. §

The following section of code lists the header of the message specified by its message number. It retrieves no body lines.

```
int l_retCode = 0;

int l_messageNumber = 1;

pop3Client_t * l_pPOP3Client = NULL;

/* Get only the message headers: request 0 body lines */

l_retCode = pop3_top(l_pPOP3Client, l_messageNumber, 0);
```

```
if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
l_retCode = pop3_processResponses(l_pPOP3Client);
if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
```

After you retrieve message headers, you can go on to Retrieving a Message if you like.

[Top]

Retrieving a Message

Retrieving a message is one of the most common activities that users want to perform during the POP3 Transaction state.

The `pop3_retrieve` function takes the identifier of the POP3 client that is retrieving the mail and and the message number, and retrieves the contents of the message:

```
int pop3_retrieve( pop3Client_t * in_pPOP3, int in_messageNumber );
```

The message is returned in the form of data chunks. The `pop3_retrieve` function issues a RETR command. It fails if the message with the specified number does not exist.

Note For this function's callback mapping, see POP3 Function Callback Mapping. §

The following section of code retrieves the contents of a message.

```
int l_retCode = 0;
int l_messageNumber = 1;
pop3Client_t * l_pPOP3Client = NULL;
l_retCode = pop3_retrieve(l_pPOP3Client, l_messageNumber);
if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
l_retCode = pop3_processResponses(l_pPOP3Client);
if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
```

[Top]

Ending the Session

When the client wants to close the session, the messaging application must deallocate the response sink, free the client, and close the socket connection with the server. For these operations, see Connecting to a Server, Creating a Client, and Creating a Response Sink.

The client calls `pop3_quit` to notify the server that it is closing the session:

```
int pop3_quit( pop3Client_t * in_pPOP3 );
```

This function sends the QUIT POP3 protocol command. The server closes the TCP connection and sends back a response. It is preferable to end a session with `pop3_quit` instead of just closing.

If the session is in the Authentication state when this function is called, the server simply closes the connection. If the session is in the Transaction state, the server goes into the Update state and expunges any messages marked for deletion, and then quits.

Note For this function's callback mapping, see POP3 Function Callback Mapping. §

The following section of code notifies the server that the client is terminating the session.

```
int l_retCode = 0;

pop3Client_t * l_pPOP3Client = NULL;

l_retCode = pop3_quit(l_pPOP3Client);

if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }

l_retCode = pop3_processResponses(l_pPOP3Client);

if (l_retCode != NSMAIL_OK) { /*Deal with error*/ }
```

[Top]

POP3 Functions by Task

To help you find the information you need more quickly, look for the task you want to perform in the column on the left. Then you can click the function name for further information.

Table 5.5 POP3 Functions by Task

Task	Function for the task
Connect to server.	pop3_connect
Mark a message for deletion on the server.	pop3_delete
Disconnect from server.	pop3_disconnect
Free the POP3 client structure and its data members.	pop3_free
Get the IO model.	pop3_get_option
Initialize and allocate the POP3 client structure and set the sink.	pop3_initialize
List all messages.	pop3_list
List the specified message.	pop3_listA
Contact the server, which sends a response indicating it is still present.	pop3_noop
Identify user password and enter the Transaction state.	pop3_pass
Process the server responses for API commands.	pop3_processResponses
Close the connection with the server.	pop3_quit
Undelete any messages marked as deleted.	pop3_reset
Retrieve message with specified number.	pop3_retrieve
Send an unsupported command to the server.	pop3_sendCommand
Set the size of the message data chunk passed to the user.	pop3_setChunkSize
Set the IO models.	pop3_set_option
Set a new response sink.	pop3_setResponseSink
Set the amount of time allowed to wait before returning control to the user.	pop3_setTimeout
Get the status of the mail drop.	pop3_stat

Task	Function for the task
Retrieve the headers plus a specified number of lines from the message.	pop3_top
Get the message numbers and the corresponding unique ids of the messages in the maildrop.	pop3_uidList
Get the specified message number and its unique id.	pop3_uidListA
Enter a user name.	pop3_user
Get a list of authenticated users.	pop3_xAuthList
Get the authenticated user for a given message.	pop3_xAuthListA
Get the email address of the sender of the specified message.	pop3_xSender
Free the POP3 response sink and its data members.	pop3Sink_free
Initialize and allocate the POP3 response sink structure.	pop3Sink_initialize

[Top] [POP3 Functions by Task]

2

Messaging Access SDK C

Reference

Chapter 8 Shared C Definitions

This chapter describes the C language functions, structures, and definitions shared by all Messaging Access SDK APIs.

Chapter 9 SMTP C API Reference

This chapter describes the functions and structures of the C language API for the SMTP (Simple Mail Transfer Protocol) protocol of the Messaging Access SDK.

Chapter 10 MIME C API Reference

This chapter describes the functions and structures of the C language API for the MIME (Multipurpose Internet Mail Extension) protocol of the Messaging Access SDK.

Chapter 11 IMAP C API Reference

This chapter describes the functions and structures of the C language API for the IMAP4 (Internet Message Access Protocol, Version 4) protocol of the Messaging Access SDK.

Chapter 12 POP3 C API Reference

This chapter describes the functions and structures of the C language API for the POP3 (Post Office Protocol) protocol of the Messaging Access SDK.

[Top]

Chapter

6

Messaging Access SDK C Reference Overview

This page contains links to the reference to the C versions of the Messaging Access SDK Protocol APIs.

Program elements are organized into Functions, Structures, and Definitions, and by name within each division. You can find functions alphabetically or by functional group.

[Top]

Chapter

7

Reference to Protocols

This chapter summarizes essential information about the Internet Protocols accessed through the Messaging Access SDK.

Messaging Access SDK Protocol APIs are based on the standard Internet messaging protocols, SMTP, IMAP4, POP3, and MIME. The SDK implementations of the protocols contain functions that call Internet Protocol commands. This chapter lists the Internet Protocol commands supported by the Messaging Access SDK, defines them, and lists the SDK functions that call them.

- Supported SMTP Internet Protocol Commands. SMTP (Simple Mail Transfer Protocol) sends messages.
- Supported IMAP4 Internet Protocol Commands. IMAP4 (Internet Message Access Protocol) retrieves and manages messages remotely.
- Supported POP3 Internet Protocol Commands. POP3 (Post Office Protocol) downloads messages to a client and allows for search and retrieval of messages.

- MIME, Multipurpose Internet Mail Extension. MIME builds code messages with attachments for sending with SMTP, and parses and decodes received messages. SDK MIME methods and functions do not map to a set of Internet Protocol commands in the same way that the other protocols do.

[Top]

Supported SMTP Internet Protocol Commands

This table lists supported protocol commands for SMTP (Simple Mail Transfer Protocol) and the SDK function that calls each command. For the RFC sources for these protocol APIs, see SMTP RFCs.

Supported Internet Protocol command	What the command does	SDK function that calls the command
DATA	Informs server that the client is about to send the message. After server OK, client sends RFC 822-compliant message data line by line. On completion, client sends "<CRLF>.<CRLF>" line.	smtp_data
DSN [NOTIFY, RET, ENVID]	Allows SMTP client to generate delivery status notifications (DSNs) when needed, determine whether the notifications return the message contents, and get additional information with a DSN so that the sender can identify both recipient(s) for the DSN and the transaction that contained the original message. (RFC 1891)	
EHLO	Client starts SMTP session by sending identification to server; server responds (identifies itself) in a greeting message. EHLO replaces older HELO command for SMTP clients that support SMTP service extensions. Can be issued at the start of the session to see which extensions the server supports.	smtp_ehlo
EXPN	Expands a mailing list alias; the command retrieves the alias member list.	smtp_expand

Supported Internet Protocol command	What the command does	SDK function that calls the command
HELP	Calls the server Help utility. Application-specific; usually lists available commands.	smtp_help
MAIL FROM	Initiates sending the message; supplies the message's reverse path (usually the sender's fully qualified domain name).	smtp_mailFrom
NOOP	Gets positive server response.	smtp_noop
QUIT	Sent by a client to a server when the client is ready to end the session. Server sends response and closes TCP connection. Best to use this command rather than just closing the connection.	smtp_quit
PIPELINING	Allows command pipelining (batching multiple commands into single TCP sends). To find out if server supports pipelining, issue the EHLO command. If it does, the server response includes code 250 and EHLO keyword PIPELINING. Pipelining allows the client to transmit batches of SMTP commands without waiting for a response to each. (RFC 2197)	smtp_setPipelining
RCPT TO	Specifies the address of a message recipient. Called once for each recipient. Follows the MAIL command.	smtp_rcptTo
RSET	Cancels the current mail transfer and all current processes, discards data, and clears session states. Returns to the session state that followed the EHLO command.	smtp_reset
VRFY	Sent by a client to verify a user name with the server. The server responds with a positive or negative code.	smtp_verify

[Top]

Supported IMAP4 Internet Protocol Commands

This table lists supported protocol commands for IMAP4 (Internet Message Access Protocol) and the SDK function that calls each command. For the RFC sources for these protocol APIs, see IMAP4 RFCs.

Supported Internet Protocol command	What the command does	Session state for command	SDK function that calls the command
APPEND	Appends message to specified mailbox, passes on any message flags.	Authenticated	imap4_append
CAPABILITY	Gets a list of server capabilities.	All states	imap4_capability
CHECK	Requests a checkpoint of the currently selected mailbox; server flushes existing mailbox states to disk.	Selected	imap4_check
CLOSE	Closes a mailbox, deletes flagged messages, moves session to Non-Authenticated state.	Selected	imap4_close
COPY	Copies a message to the specified mailbox.	Selected	imap4_copy
CREATE	Creates a mailbox.	Authenticated	imap4_create
DELETE	Marks a message for deletion.	Authenticated	imap4_delete
EXAMINE	Like SELECT, but read-only.	Authenticated	imap4_examine
EXPUNGE	Removes all messages flagged "\\Deleted" in a mailbox.	Selected	imap4_expunge
FETCH	Returns information from messages. The protocol's FETCH [XSENDER] form is supported.	Selected	imap4_fetch
LIST	Gets list of user names.	Authenticated	imap4_list
LOGIN	Logs in to server with user name and password.	Non-Authenticated	imap4_login
LOGOUT	Ends session; server responds with "BYE."	All states	imap4_logout

Supported Internet Protocol command	What the command does	Session state for command	SDK function that calls the command
LSUB	Lists members of subscription list (must be added with SUBSCRIBE).	Authenticated	imap4_lsub
NAMESPACE	Retrieves the prefixes of namespaces used by a server for personal mailboxes, other user's mailboxes, and shared mailboxes.	All states	imap4_namespace
NOOP	Gets positive server response.	All states	imap4_noop
RENAME	Renames mailbox.	Authenticated	imap4_rename
SEARCH	Finds messages that meet specified criteria.	Selected	imap4_search
SELECT	Selects a mailbox on the server for operations involving messages.	Authenticated	imap4_select
STATUS	Requests one or more types of status for the specified mailbox.	Authenticated	imap4_status
STORE	Updates flags on messages; can return the new flag status.	Selected	imap4_store
SUBSCRIBE	Adds a mailbox to a server's subscribed mailbox list. List is accessed with LSUB.	Authenticated	imap4_subscribe
UID	Used with a command name to specify that it uses unique message identifiers.	Selected	imap4_uidCopy imap4_uidFetch imap4_uidSearch imap4_uidStore
UNSUBSCRIBE	Deletes a mailbox from a server's subscribed mailbox list. List is accessed with LSUB.	Authenticated	imap4_unsubscribe
SETACL	Changes the access control list on the specified mailbox and grants specified permissions.	All states	imap4_setACL
DELETEACL	Removes an <identifier, rights> pair for the specified identifier from the access control list for the specified mailbox.	All states	imap4_deleteACL
GETACL	Retrieves the access control list for the mailbox in an untagged ACL reply.	All states	imap4_getACL

Supported Internet Protocol command	What the command does	Session state for command	SDK function that calls the command
LISTRIGHTS	Retrieves the access control list for mailbox in an untagged ACL reply.	All states	imap4_listRights
MYRIGHTS	Retrieves the user's rights to the specified mailbox.	All states	imap4_myRights

[Top]

Supported POP3 Internet Protocol Commands

This table lists supported protocol commands for POP3 (Post Office Protocol) and the SDK function that calls each command. For the RFC sources for these protocol APIs, see POP3 RFCs.

Supported Internet Protocol command	What the command does	Session state for command	SDK function that calls the command
DELE	Asks server to mark specified message for deletion. Deletion actually takes place on entry into Update state.	Transaction	pop3_delete
LIST	Gets the size of one or all messages.	Transaction	pop3_list pop3_listA
NOOP	Gets positive server response.	Transaction	pop3_noop
PASS	Identifies a user password; on success, moves session to the Transaction state.	Authorization	pop3_pass
QUIT	Ends the session. If issued in Authentication state, server closes connections. If issued in Transaction state, server goes into Update and deletes any marked messages, then quits.	Authorization, Update	pop3_quit

Supported Internet Protocol command	What the command does	Session state for command	SDK function that calls the command
RSET	Asks the server to clear all delete tags from messages.	Transaction	pop3_reset
RETR	Requests the entire specified message.	Transaction	pop3_retrieve
STAT	Gets the number of messages in and octet size of mail drop.	Transaction	pop3_stat
TOP	Asks server for first *n* lines of specified message.	Transaction	pop3_top
UIDL	Gets the unique identifier string for specified or all messages.	Transaction	pop3_uidList pop3_uidListA
USER	Identifies the user or mail drop by name to the server; the server returns a known or unknown response.	Authorization	pop3_user
XAUTHLIST	Returns a list of authenticated users.	Transaction	pop3_xAuthList pop3_xAuthListA
XSENDER	Gets the email address of the sender of the specified message. Client uses this to query whether an individual message has been authenticated. Server returns an empty OK string if no authenticated sender is found.	Transaction	pop3_xSender

[Top]

Chapter

8

Shared C Definitions

This chapter describes the C language functions, structures, and definitions shared by all Messaging Access SDK APIs.

Definitions that apply only to a particular protocol are included in the reference chapter for that protocol.

- Shared Definitions
- Shared Structures
- Shared Functions

All shared functions, structures, and definitions are found in two header files:

- The `nsmail.h` header file contains definitions, structures, and functions, used by the Messaging Access SDK Protocol APIs.
- The `nsStream.h` header file contains definitions for I/O streaming utilities.

[Top]

Shared Definitions

The SDK Protocol APIs share a set of definitions.

- Constants
- Option Definition
- Error Definitions

[Top]

Constants

These constants are defined as `unsigned char boolean`.

Definition	Value	Description
FALSE	0	`unsigned char boolean`
TRUE	1	`unsigned char boolean`
BOOLEAN	`true`/`false`	`int`; Unix

[Top] [Shared Definitions]

Option Definition

This is the definition for options that you might want to set by using the SMTP, IMAP4, and POP3 `set_option` functions. Only one option is supported.

Option	Definition
`NSMAIL_OPT_IO_FN_PTRS`	Option for setting specific I/O functionality. See `nsmail_io_fns_t`.

[Top] [Shared Definitions]

Error Definitions

Error	Value	Definition
`NSMAIL_OK`	0	Successful completion of function.
`NSMAIL_ERR_UNINTIALIZED`	-1	Uninitialized value.
`NSMAIL_ERR_INVALIDPARAM`	-2	Invalid parameters.
`NSMAIL_ERR_OUTOFMEMORY`	-3	Out of memory.
`NSMAIL_ERR_UNEXPECTED`	-4	Unexpected element.
`NSMAIL_ERR_IO`	-5	Error in input/output operation.
`NSMAIL_ERR_IO_READ`	-6	Error in reading input stream.
`NSMAIL_ERR_IO_WRITE`	-7	Error in writing to output stream.
`NSMAIL_ERR_IO_SOCKET`	-8	Socket connection error.
`NSMAIL_ERR_IO_SELECT`	-9	Error in selecting message.
`NSMAIL_ERR_IO_CONNECT`	-10	Error in connecting.
`NSMAIL_ERR_IO_CLOSE`	-11	Error in closing.
`NSMAIL_ERR_PARSE`	-12	Internal parsing error occurred.

Error	Value	Definition
NSMAIL_ERR_TIMEOUT	-13	Timeout occurred. Recoverable error. Wait and call processResponses later.
NSMAIL_ERR_INVALID_INDEX	-14	Index is invalid.
NSMAIL_ERR_CANTOPENFILE	-15	Cannot open file.
NSMAIL_ERR_CANT_SET	-16	Value cannot be set.
NSMAIL_ERR_ALREADY_SET	-17	Value already set.
NSMAIL_ERR_CANT_DELETE	-18	Cannot delete item.
NSMAIL_ERR_CANT_ADD	-19	Cannot add item.
NSMAIL_ERR_SENDDATA	-20	Data send failed. Recoverable error.
NSMAIL_ERR_MUSTPROCESSRESPONSES	-21	Response not processed. Recoverable error.
NSMAIL_ERR_ PIPELININGNOTSUPPORTED	-22	Server does not support pipelining. Recoverable error.
NSMAIL_ERR_ALREADYCONNECTEDD	-23	Already connected.
NSMAIL_ERR_NOTCONNECTED	-24	Not connected.
NSMAIL_ERR_EOF	-1	End of file reached.
NSMAIL_ERR_NOTIMPL	-99	Function or feature not implemented.

[Top] [Shared Definitions]

Shared Structures

This section defines Messaging Access SDK data structures, listed in alphabetical order. These structures represent input and output streams used by the SDK Protocol APIs.

```
nsmail_io_fns_t
nsmail_inputstream_t
nsmail_outputstream_t
```

[Top]

---

nsmail_io_fns_t

Definition for a structure to plug in specific IO functionality.

Syntax

```
include <nsmail.h>
typedef struct nsmail_io_fns
{
   int (*liof_read)(int, char *, int);
   int (*liof_write)(int, char *, int);
   int (*liof_socket)(int, int, int);
   int (*liof_select)(int, fd_set *,
         fd_set *, fd_set *, struct timeval *);
   int (*liof_connect)(int,
         const struct sockaddr*, int);
   int (*liof_close)(int);
} nsmail_io_fns_t;
```

Fields The structure has the following fields:

`int (*liof_read)(int, char *, int);`	Reads data.
`int (*liof_write)(int, char *, int);`	Writes data.
`int (*liof_socket)(int, int, int);`	Creates the socket handle.
`int (*liof_select)(int, fd_set *, fd_set *, fd_set *, struct timeval *);`	Selects.
`int (*liof_connect)(int, const struct sockaddr*, int);`	Eastablishes a connection.
`int (*liof_close)(int);`	Closes the connection.

Description This structure is made up of pointers to the standard functions supported on most platforms. In order to plug in specific input and output functionality, the developer must define these functions.

See Also smtp set option, imap4 set option, pop3 set option

[Top] [Shared Structures]

---

nsmail_inputstream_t

Definition for a structure to plug in specific input streaming functionality.

Syntax

```
include <nsmail.h>
typedef struct nsmail_inputstream

   void *rock;
   int (*read) (void *rock,
         char *buf, unsigned size);
   void (*rewind) (void *rock);
   void (*close) (void *rock);
} nsmail_inputstream_t;
```

Fields The structure has the following fields:

`void *rock`	Opaque client data.
`int (*read) ( void *rock, char *buf, unsigned size);`	Returns the number of bytes actually read and -1 on eof on the stream; may return an `NSMAIL_ERR* (int value < 0)` in case of any other error. Can be invoked multiple times. Each read returns the next sequence of bytes in the stream, starting after the last byte returned by the previous read. If the number of bytes actually returned by read is less than `size` parameter, an eof on the stream is assumed. Parameters: • `buf`: space allocated and freed by the caller. • `size:` maximum number of bytes to be returned by read. If the number of bytes returned by read is less than `size`, an eof on the stream is assumed.
`void (*rewind) (void *rock);`	Reset the stream to the beginning. A subsequent read returns data from the start of the stream.
`void (*close) (void *rock);`	Closes the stream, freeing any internal buffers and `rock` parameter. Once a stream is closed, it is illegal to attempt `read`, `rewind`, or `close` on the stream. After `close`, the user of the stream must free the `nsmail_inputstream` structure corresponding to the stream.

Description This structure is made up of pointers to functions. The developer must define these functions.

See Also `nsmail_outputstream_t`, `smtp sendStream`

[Top] [Shared Structures]

---

nsmail_outputstream_t

Definition for a structure to plug in specific output streaming functionality.

Syntax

```
include <nsmail.h>
typedef struct nsmail_outputstream
{
   void *rock;
   void (*write) (void *rock, const char *buf,
```

```
        unsigned size);
   void (*close) (void *rock);
} nsmail_outputstream_t;
```

Fields The structure has the following fields:

`void *rock;`	Opaque client data.
`void (*write) (void *rock, const char *buf, unsigned size);`	Writes CRLF-separated output in `buf` parameter, of size `size`. May be called multiple times.
`void (*close) (void *rock);`	Closes the stream, freeing any internal buffers, rock, and other data. After a stream is closed, it is illegal to attempt `write` or `close` operations on the stream. After `close`, the user of the stream must free the `nsmail_outputstream` structure corresponding to the stream.

Description This structure is made up of pointers to functions. The developer must define these functions.

See Also `nsmail_inputstream_t`

[Top] [Shared Structures]

Shared Functions

The Messaging Access SDK defines several functions that create input and output streams used by the SDK Protocol APIs.

```
file_inputStream_create
buf_inputStream_create
file_outputStream_create
nsStream_free
```

To use the functions declared by this header file, link with the `libcomm.so` file.

[Top]

file_inputStream_create

Creates an input stream from a file.

Syntax

```
#include <nsStream.h>
int file_inputStream_create (char * in_fileName,
      nsmail_inputstream_t ** out_ppRetInputStream);
```

Parameters The function has the following parameters:

`in_fileName`	Full path name of an existing file on which to create the input stream.
`out_ppRetInputStream`	The created input stream is returned here.

Returns

- If successful, the function returns `NSMAIL_OK` (0).
- If unsuccessful, the function returns an error message (< 0). For a complete list, see Error Definitions.

Description This function creates an input stream that can be passed to all Messaging Access SDK API calls that require a parameter of type `nsmail_inputstream_t`. The function implements the stream and the associated internal functions for `read`, `rewind` and `close`.

See Also `buf_inputStream_create`

Example This code shows how to use this function.

```
nsmail_inputstream_t * pInputStream;

ret_value = file_inputStream_create (
            <file-name>, &pInputStream);

ret_value = mime_basicPart_setDataStream (
            BasicPart, pInputStream, boolean);

/* When done, close and free the stream. */

pInputStream->close (pInputStream->rock);

free (pInputStream); pInputStream = NULL;
```

[Top] [Shared Functions]

buf_inputStream_create

Creates an input stream from a buffer.

Syntax

```
#include <nsStream.h>
int buf_inputStream_create (
         char * in_pDataBuf,
         long in_data_size,
         nsmail_inputstream_t ** out_ppRetInputStream);
```

Parameters The function has the following parameters:

`in_pDataBuf`	Data buffer on which to base the input stream. Should not be `null`. A reference to the buffer is stored in the stream. Do not free the buffer until the stream is closed.
`in_data_size`	Size of data in `in_pDataBuf`.
`out_ppRetInputStream`	The created input stream is returned here.

Returns

- If successful, the function returns `NSMAIL_OK` (0).
- If unsuccessful, the function returns an error message (< 0). For a complete list, see Error Definitions.

Description This function creates an input stream that can be passed to all Messaging Access SDK API calls that require a parameter of type `nsmail_inputstream_t`. The function implements the stream and the associated internal functions for `read`, `rewind` and `close`.

Example This function is used in the same way shown in the Example for `file_inputStream_create`.

See Also `file_inputStream_create`

[Top] [Shared Functions]

file_outputStream_create

Creates an input stream from a buffer.

Syntax

```
#include <nsStream.h>
int file_outputStream_create (
      char * in_fileName,
      nsmail_outputstream_t ** out_ppRetOutputStream);
```

Parameters The function has the following parameters:

`in_fileName`	Full path name of the file on which to create the output stream. Creates the file if needed. Deletes any previous data in the file.
`out_ppRetOutputStream`	The created output stream is returned here.

Returns

- If successful, the function returns `NSMAIL_OK` (0).
- If unsuccessful, the function returns an error message (< 0). For a complete list, see Error Definitions.

Description This function creates an output stream that can be passed to all Messaging Access SDK API calls that require a parameter of type `nsmail_outputstream_t`.

See Also `file_inputStream_create, buf_inputStream_create`

Example This code shows how to use this function. The file under the stream must be removed separately.

```
nsmail_outputstream_t * pOutputStream;

ret_value = file_outputStream_create (<file-name>,
              &pOutputStream);

ret_value = mime_message_putByteStream (
              <other-params>, pOutputStream);

/* When done, close and free the stream. */

pOutputStream->close (pOutputStream->rock);

nsStream_free (pOutputStream);

pOutputStream = NULL;
```

[Top] [Shared Functions]

nsStream_free

Frees the memory allocated for the input or output stream.

Syntax

```
#include <nsStream.h>
void nsStream_free (void * pMem);
```

Parameters

The function has the following parameters:

`pMem`	Stream to free.

Returns

- If successful, the function returns `NSMAIL_OK` (0).
- If unsuccessful, the function returns an error message (< 0). For a complete list, see Error Definitions.

See Also

`file_inputStream_create, buf_inputStream_create, file_outputStream_create`

[Top] [Shared Functions]

Chapter

9

SMTP C API Reference

This chapter describes the functions and structures of the C language API for the SMTP (Simple Mail Transfer Protocol) protocol of the Messaging Access SDK.

- SMTP Functions
- SMTP Structures

You'll find links to each function and structure in this introduction. Each reference entry gives the name of the function or structure, its header file, its syntax, and its parameters.

All C interface definitions are found in the `smtp.h` file.

[Top]

SMTP Functions

This table lists SMTP functions alphabetically by name, with descriptions. Click the function name to get information about it.

Function	Description
smtp_bdat	Sends binary data chunks to the server.
smtp_connect	Connects to the server using the default port.
smtp_data	Prepares to send data to the server.
smtp_disconnect	Closes the socket connection with the server.
smtp_ehlo	Sends the EHLO command to the server.
smtp_expand	Expands a given mailing list.
smtp_free	Frees the smtpClient_t structure.
smtp_get_option	Gets IO models.
smtp_help	Gets help on a given topic.
smtp_initialize	Initializes and allocates the smtpClient_t structure and sets the response sink.
smtp_mailFrom	Initiates sending message; supplies message's reverse path.
smtp_noop	Gets positive server response; does not affect SMTP session.
smtp_processResponses	Processes the server responses for API commands.
smtp_quit	Closes the connection with the server.
smtp_rcptTo	Specifies a message recipient's address.
smtp_reset	Resets the state of the server; discards any sender and recipient information.
smtp_send	Sends message data to the server.
smtp_sendCommand	Sends an unsupported command to the server.
smtp_sendStream	Sends message data from a stream to the server.
smtp_setChunkSize	Sets the size of the data chunks to send.
smtp_set_option	Sets IO models.
smtp_setPipelining	Attempts to enable pipelining.

Function	Description
smtp_setResponseSink	Sets a new response sink.
smtp_setTimeout	Sets the amount of time allowed to wait before returning control to the user.
smtp_verify	Verifies a username.
smtpSink_free	Frees the SMTP response sink and its data members.
smtpSink_initialize	Initializes and allocates the smtpSink_t structure.

[Top] [SMTP Functions] [SMTP Functions by Task]

smtp_bdat

Sends binary data chunks to the server.

Syntax

```
#include <smtp.h>
int smtp_bdat( smtpClient_t * in_pSMTP,
            const char * in_data,
            unsigned int length,
            boolean in_fLast);
```

Parameters The function has the following parameters:

in_pSMTP	Pointer to the SMTP client.
in_data	Pointer to message data.
length	Size of the binary chunk to send. Default: 1K.
in_fLast	Whether this is the last data chunk to send. Values: • true: last chunk. • false: more data to come.

Returns

- If successful, the function returns NSMAIL_OK.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function sends data in binary chunks of the size you determine. Call `smtp_bdat` multiple times until the boolean value in the `in_fLast` parameter is set to `true`.

If you use the `smtp_send` or `smtp_sendStream` function, send data with `smtp_data` and not with `smtp_bdat`. For more information, see Sending the Message.

This function sends the DATA SMTP protocol command.

Note `bdat` is not supported by Messaging Server 4.0. Use `smtp_data` instead. §

This function is mapped to one or more callbacks in the `smtpSink_t` structure. For more information, see SMTP Function Callback Mapping.

See Also `smtpSink_t.bdat`, `smtpSink_t.error`, `smtp_data`, `smtp_sendStream`, `smtp_send`, `smtp_setChunkSize`

[Top] [SMTP Functions] [SMTP Functions by Task]

---

smtp_connect

Connects to the server using the specified port.

Syntax

```
#include <smtp.h>
int smtp_connect( smtpClient_t * in_pSMTP,
             const char * in_server,
             unsigned short in_port );
```

Parameters The function has the following parameters:

`in_pSMTP`	Pointer to the SMTP client.
`in_server`	Pointer to the name of the server.
`in_port`	Server port to connect to. A value of 0 connects using the Protocol standard default port (25).

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description For more information, see Connecting to a Server.

This function is mapped to one or more callbacks in the `smtpSink_t` structure. For more information, see SMTP Function Callback Mapping.

See Also `smtpSink_t.connect`, `smtpSink_t.error`, `smtp_quit`, `smtp_ehlo`

[Top] [SMTP Functions] [SMTP Functions by Task]

---

smtp_data

Prepares to send data to the server.

Syntax

```
#include <smtp.h>
int smtp_data(smtpClient_t * in_pSMTP);
```

Parameters The function has the following parameters:

`in_pSMTP`	Pointer to the SMTP client.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function sends data in conjunction with `smtp_send` or `smtp_sendstream`. Call `smtp_data`, then call one of these functions. You must always use `smtp_data` and not `smtp_bdat` with either of these functions. This function sends the DATA SMTP protocol command. For more information, see Sending the Message.

This function is mapped to one or more callbacks in the `smtpSink_t` structure. For more information, see SMTP Function Callback Mapping.

See Also `smtpSink_t.data`, `smtpSink_t.error`, `smtp_bdat`, `smtp_send`, `smtp_sendStream`

[Top] [SMTP Functions] [SMTP Functions by Task]

smtp_disconnect

Closes the socket connection with the server.

Syntax

```
#include <smtp.h>
int smtp_disconnect( smtpClient_t * in_pSMTP );
```

Parameters The function has the following parameters:

`in_pSMTP`	Pointer to the SMTP client to disconnect.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function closes the socket connection. It could be used to implement a Cancel button.

Note Do not call the `SMTP_processResponses` function after this function. Operations for which you do not call `processResponses`: disconnecting, the set functions, initializing and freeing the client and sink. §

See Also `smtp_connect, smtp_ehlo`

[Top] [SMTP Functions] [SMTP Functions by Task]

smtp_ehlo

Starts the SMTP session

Syntax

```
#include <smtp.h>
int smtp_ehlo( smtpClient_t * in_pSMTP,
            const char * in_domain );
```

Parameters The function has the following parameters:

`in_pSMTP`	Pointer to the SMTP client.
`in_domain`	Pointer to the server domain name string.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function returns a multiline message listing the SMTP extensions, such as pipelining or DSN, that are supported by the server. This is similar to the functionality of the IMAP4 `capability` command. This function sends the EHLO SMTP protocol command. For more information, see Determining ESMTP Support.

This function is mapped to one or more callbacks in the `smtpSink_t` structure. For more information, see SMTP Function Callback Mapping.

See Also `smtpSink_t.ehlo`, `smtpSink_t.error`, `smtpSink_t.ehloComplete`, `smtp_connect`, `imap4_capability`

[Top] [SMTP Functions] [SMTP Functions by Task]

---

smtp_expand

Expands the specified mailing list.

Syntax

```
#include <smtp.h>
int smtp_expand(smtpClient_t * in_pSMTP,
             const char * in_mailingList);
```

Parameters The function has the following parameters:

`in_pSMTP`	Pointer to the SMTP client.
`in_mailingList`	Pointer to name of user mailing list you want to expand.

Returns

- If successful, the function returns `NSMAIL_OK`.

- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function gets the email addresses of the users on the mailing list. It sends the EXPN SMTP protocol command.

This function is mapped to one or more callbacks in the `smtpSink_t` structure. For more information, see SMTP Function Callback Mapping.

See Also `smtpSink_t.expand`, `smtpSink_t.expandComplete`, `smtpSink_t.error`

[Top] [SMTP Functions] [SMTP Functions by Task]

---

smtp_free

Frees the `SMTPClient_t` structure.

Syntax

```
#include <smtp.h>
void smtp_free( smtpClient_t ** in_ppSMTP );
```

Parameters The function has the following parameters:

`in_ppSMTP`	Pointer to a pointer to a client structure.

Description This function frees the `SMTPClient_t` structure. When the function returns, the client structure is set to `null`. The user must free any pointers to opaque data.

See Also `smtp_initialize`, `smtpSink_free`

[Top] [SMTP Functions] [SMTP Functions by Task]

---

smtp_get_option

Gets IO models.

Syntax

```
#include <smtp.h>
int smtp_get_option( smtpClient_t * in_pSMTP,
                     int in_option,
                     void * in_pOptionData );
```

Parameters The function has the following parameters:

`in_pSMTP`	Pointer to the SMTP client.
`in_option`	Pointer to option you want to set. See `nsmail_io_fns_t`. For the option that is currently supported, see "Option Definition."
`in_pOptionData`	Pointer to option data set with `smtp_set_option`.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This Do not call `processResponses` after this function.

See Also `smtp_set_option`

[Top] [SMTP Functions] [SMTP Functions by Task]

---

smtp_help

Asks the server for help on a specified topic.

Syntax

```
#include <smtp.h>
int smtp_help( smtpClient_t * in_pSMTP,
               const char * in_helpTopic);
```

Parameters The function has the following parameters:

`in_pSMTP`	Pointer to the SMTP client.
`in_helpTopic`	Pointer to one-word help topic name. If `null`, displays help information on all supported commands.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function asks the server for help on a specified topic. It sends the HELP SMTP protocol command.

This function is mapped to one or more callbacks in the `smtpSink_t` structure. For more information, see SMTP Function Callback Mapping.

See Also `smtpSink_t.help`, `smtpSink_t.helpComplete`, `smtpSink_t.error`

[Top] [SMTP Functions] [SMTP Functions by Task]

---

smtp_initialize

Initializes and allocates the `SMTPClient_t` structure and sets the response sink.

Syntax

```
#include <smtp.h>
int smtp_initialize(smtpClient_t ** out_ppSMTP,
               smtpSink_t * in_psmtpSink );
```

Parameters The function has the following parameters:

`out_ppSMTP`	Pointer to a pointer to an SMTP client structure.
`in_psmtpSink`	Pointer to the response sink structure to use.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description Do not call `processResponses` after this function.

For more information, see Creating a Client.

See Also `smtp_connect, smtpSink_free`

[Top] [SMTP Functions] [SMTP Functions by Task]

---

smtp_mailFrom

Initiates sending message; supplies message's reverse path.

Syntax

```
#include <smtp.h>
int smtp_mailFrom( smtpClient_t * in_pSMTP,
             const char * in_reverseAddress,
             const char * in_esmtpParams );
```

Parameters The function has the following parameters:

`in_pSMTP`	Pointer to the SMTP client.
`in_reverseAddress`	Message's reverse path, usually sender's fully qualified domain name.
`in_esmtpParams`	Pointer to optional ESMTP (Extended SMTP) parameters.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function initiates sending the message. The `in_reverseAddress` parameter supplies the message's reverse path, usually the sender's fully qualified domain name. This function sends the MAIL FROM: SMTP protocol command. For more information, see Setting the Mailer.

Note Using the ESMTP option DSN requires a detailed understanding of the SMTP protocol. (Links to RFCs: SMTP RFCs) §

This function is mapped to one or more callbacks in the `smtpSink_t` structure. For more information, see SMTP Function Callback Mapping.

See Also `smtpSink_t.mailFrom`, `smtpSink_t.error`, `smtp_rcptTo`

[Top] [SMTP Functions] [SMTP Functions by Task]

---

smtp_noop

Gets positive server response.

Syntax

```
#include <smtp.h>
int smtp_noop(smtpClient_t * in_pSMTP);
```

Parameters The function has the following parameters:

`in_pSMTP`	Pointer to the SMTP client.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description If you send any command, the server responds with a "still here" response. Sending the `smtp_noop` function does nothing except force this server response. You can use this function to maintain server connection, perhaps issuing it at timed intervals to make sure that the server is still active.

Your application may not need this function. For example, if the application does something and then disconnects at once, there is no need to make sure that the server is still connected.

This function sends the NOOP SMTP protocol command. It is mapped to one or more callbacks in the `smtpSink_t` structure. For more information, see SMTP Function Callback Mapping.

See Also `smtpSink_t.noop`, `smtpSink_t.error`

[Top] [SMTP Functions] [SMTP Functions by Task]

---

smtp_processResponses

Processes the server responses for API commands.

Syntax

```
#include <smtp.h>
int smtp_processResponses( smtpClient_t * in_pSMTP );
```

Parameters The function has the following parameters:

`in_pSMTP`	Pointer to the SMTP client.

Returns

- If successful, the function returns `NSMAIL_OK`.

- If a time-out occurs, the function returns.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function reads in responses from the server. It invokes the callback functions provided by the user for all responses that are available at the time of execution.

If a time-out occurs, the user can continue by calling `smtp_processResponses` again.

Do not call `SMTP_processResponses` after these operations: disconnecting, using any of the set functions, or initializing or freeing either the client or the response sink.

See Also `smtpSink_t`

[Top] [SMTP Functions] [SMTP Functions by Task]

---

smtp_quit

Closes the connection with the server.

Syntax

```
#include <smtp.h>
int smtp_quit(smtpClient_t * in_pSMTP);
```

Parameters The function has the following parameters:

`in_pSMTP`	Pointer to the SMTP client.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function sends the QUIT SMTP protocol command. Issuing this function ends the session. For more information, see Ending the Session.

This function is mapped to one or more callbacks in the `smtpSink_t` structure. For more information, see SMTP Function Callback Mapping.

See Also smtpSink_t.quit, smtpSink_t.error

[Top] [SMTP Functions] [SMTP Functions by Task]

---

smtp_rcptTo

Specifies a message recipient's address.

Syntax

```
#include <smtp.h>
int smtp_rcptTo( smtpClient_t * in_pSMTP,
             const char * in_forwardAddress,
             const char * in_esmtpParams );
```

Parameters The function has the following parameters:

in_pSMTP	Pointer to the SMTP client.
in_forwardAddress	Pointer to the recipient's address.
in_esmtpParams	Pointer to optional ESMTP (Extended SMTP) parameters.

Returns

- If successful, the function returns NSMAIL_OK.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function specifies the address of a message recipient. It is called once for each recipient and should follow the smtp_mailFrom command. For more information, see Setting the Recipient.

This function sends the RCPT TO SMTP protocol command. It is mapped to one or more callbacks in the smtpSink_t structure. For more information, see SMTP Function Callback Mapping.

See Also smtpSink_t.rcptTo, smtpSink_t.error, smtp_mailFrom

[Top] [SMTP Functions] [SMTP Functions by Task]

---

smtp_reset

Resets the state of the server; discards any sender and recipient information.

Syntax

```
#include <smtp.h>
int smtp_reset( smtpClient_t * in_pSMTP );
```

Parameters The function has the following parameters:

`in_pSMTP`	Pointer to the SMTP client.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function cancels the current mail transfer and resets the state of the server. It sends the RSET SMTP protocol command, causing the server to return to the state that followed the last function that sent the EHLO command.

This function is mapped to one or more callbacks in the `smtpSink_t` structure. For more information, see SMTP Function Callback Mapping.

See Also `smtpSink_t.reset`, `smtpSink_t.error`, `smtp_ehlo`

[Top] [SMTP Functions] [SMTP Functions by Task]

---

smtp_send

Sends message data to the server.

Syntax

```
#include <smtp.h>
int smtp_send( smtpClient_t * in_pSMTP,
               const char * in_data );
```

Parameters The function has the following parameters:

`in_pSMTP`	Pointer to the SMTP client.
`in_data`	Pointer to message data to send.

Returns

- If successful, the function returns NSMAIL_OK.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description

This function sends data in conjunction with smtp_data. Call smtp_data, then call this function. For more information, see Sending the Message.

You can use either the send or sendStream function to deliver data to the server. smtp_send sends data in a single chunk, while smtp_sendStream sends the data in a series of smaller chunks. If you use one of these functions, prepare to send data with smtp_data and not smtp_bdat.

This function is mapped to one or more callbacks in the smtpSink_t structure. For more information, see SMTP Function Callback Mapping.

See Also

smtpSink_t.send, smtpSink_t.error, smtp_bdat, smtp_data, smtp_sendStream

[Top] [SMTP Functions] [SMTP Functions by Task]

---

smtp_sendCommand

Sends an SMTP command that is not otherwise supported by the SDK.

Syntax

```
#include <smtp.h>
int smtp_sendCommand( smtpClient_t * in_pSMTP,
                const char * in_command);
```

Parameters

The function has the following parameters:

in_pSMTP	Pointer to the SMTP client.
in_command	Pointer to the unsupported command to send.

Returns

- If successful, the function returns NSMAIL_OK.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description You can use this function to extend the protocol to meet your application needs. Using this function, you can extend the functionality of the SDK to meet your needs by adding functions or passing different parameters to a function.

This function is mapped to one or more callbacks in the `smtpSink_t` structure. For more information, see SMTP Function Callback Mapping.

See Also `smtpSink_t.sendCommand`, `smtpSink_t.SendCommandComplete`, `smtpSink_t.error`

[Top] [SMTP Functions] [SMTP Functions by Task]

---

smtp_sendStream

Sends message data from a stream to the server.

Syntax

```
#include <smtp.h>
int smtp_sendStream( smtpClient_t * in_pSMTP,
                nsmail_inputstream_t * in_inputStream );
```

Parameters The function has the following parameters:

`in_pSMTP`	Pointer to the SMTP client.
`in_inputStream`	Pointer to the input stream source of the data to send.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function sends data in conjunction with `smtp_data`. Call `smtp_data`, then call this function. For more information, see Sending the Message.

You can use either the `send` or `sendStream` function to deliver data to the server. `smtp_send` sends data in a single chunk, while `smtp_sendStream` sends the data in a series of smaller chunks. If you use one of these functions, prepare to send data with `smtp_data` and not `smtp_bdat`.

This function is mapped to one or more callbacks in the `smtpSink_t` structure. For more information, see SMTP Function Callback Mapping.

See Also smtpSink_t.send, smtpSink_t.error, smtp_bdat, smtp_data, smtp_send

[Top] [SMTP Functions] [SMTP Functions by Task]

---

smtp_setChunkSize

Sets the size of the data chunks to send.

Syntax

```
#include <smtp.h>
int smtp_setChunkSize( smtpClient_t * in_pSMTP,
                int in_chunkSize );
```

Parameters The function has the following parameters:

in_pSMTP	Pointer to the SMTP client.
in_chunkSize	Size of message data chunk to set. Default: 1 K.

Returns

- If successful, the function returns NSMAIL_OK.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function sets the size of the message data chunks that are read from the input stream and sent to the server. For more information, see Sending the Message.

See Also smtp_sendStream

[Top] [SMTP Functions] [SMTP Functions by Task]

---

smtp_set_option

Sets IO models.

Syntax

```
#include <smtp.h>
```

```
int smtp_set_option( smtpClient_t * in_pSMTP,
                     int in_option,
                     void * in_pOptionData );
```

Parameters The function has the following parameters:

`in_pSMTP`	Pointer to the SMTP client.
`in_option`	Pointer to option you want to set. See `nsmail_io_fns_t`. For the option that is currently supported, see "Option Definition."
`in_pOptionData`	Pointer to option data you want to set.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description To inquire on options set with this function, use `smtp_get_option`.

Note Do not call the `processResponses` function after this function. §

See Also `smtp_get_option`

[Top] [SMTP Functions] [SMTP Functions by Task]

---

smtp_setPipelining

Attempts to enable pipelining.

Syntax

```
#include <smtp.h>
int smtp_setPipelining( smtpClient_t * in_pSMTP,
                        boolean in_enablePipelining);
```

Parameters The function has the following parameters:

`in_pSMTP`	Pointer to the SMTP client.
`in_enablePipelining`	Whether to enable pipelining. Values: • `true`: enable pipelining. • `false`: do not enable pipelining.

Returns

- If successful, the function returns NSMAIL_OK.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function attempts to enable or disable pipelining. Pipelining is enabled if the server supports it. If not, the way the network works does not change. This function sends the PIPELINING SMTP protocol command.

Pipelining is a way to batch functions. If enabled, pipelinable functions are stored locally until a specific function triggers the commands to be sent. When triggered by any non-pipelinable function, all functions begin to execute.

The pipelinable functions are `smtp_bdat`, `smtp_mailFrom`, `smtp_rcptTo`, `smtp_send`, and `smtp_sendStream`.

You can enable pipelining at any time, but it may make sense to do this immediately after invoking the `smtp_ehlo` function.

For more information, see Pipelining Commands.

Note Do not call the `processResponses` function after this function. Functions for which you do not call `processResponses`: disconnecting, the set functions, initializing and freeing the client and sink. §

See Also `smtp_ehlo`, `smtp_bdat`, `smtp_mailFrom`, `smtp_rcptTo`, `smtp_send`, `smtp_sendStream`

[Top] [SMTP Functions] [SMTP Functions by Task]

---

smtp_setResponseSink

Sets a new response sink.

Syntax

```
#include <smtp.h>
int smtp_setResponseSink ( smtpClient_t * in_pSMTP,
                      smtpSink_t * in_psmtpSink );
```

Parameters The function has the following parameters:

`in_pSMTP`	Pointer to the SMTP client.
`in_psmtpSink`	Pointer to the response sink to use.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function overrides the response sink passed into the initialized function.

Note Do not call the `processResponses` function after this function. Functions for which you do not call `processResponses`: disconnecting, the set functions, initializing and freeing the client and sink. §

See Also `smtpSink_t`

[Top] [SMTP Functions] [SMTP Functions by Task]

---

smtp_setTimeout

Sets the amount of time allowed to wait before returning control to the user.

Syntax

```
#include <smtp.h>
int smtp_setTimeout( smtpClient_t * in_pSMTP,
                double in_timeout );
```

Parameters The function has the following parameters:

`in_pSMTP`	Pointer to the SMTP client.
`in_timeout`	Time-out period to set in seconds. Values, in seconds: • -1 = infinite time-out (default) • 0 = no waiting • >0 = length of time-out period

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function sets the amount of time, in seconds, allowed to wait before returning control to the user.

Note Do not call the `processResponses` function after this function. Functions for which you do not call `processResponses`: disconnecting, the set functions, initializing and freeing the client and sink. §

See Also `smtp_processResponses`

[Top] [SMTP Functions] [SMTP Functions by Task]

---

smtp_verify

Verifies a user name.

Syntax

```
#include <smtp.h>
int smtp_verify( smtpClient_t * in_pSMTP,
             const char * in_user );
```

Parameters The function has the following parameters:

`in_pSMTP`	Pointer to the SMTP client.
`in_user`	Pointer to the user name to verify.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function sends the VRFY SMTP protocol command. It is mapped to one or more callbacks in the `smtpSink_t` structure. For more information, see SMTP Function Callback Mapping.

See Also `smtpSink_t.verify`, `smtpSink_t.error`

[Top] [SMTP Functions] [SMTP Functions by Task]

---

smtpSink_free

Frees the SMTP response sink and its data members.

Syntax

```
#include <pop3.h>
void smtpSink_free( smtpSink_t ** in_ppsmtp_Sink );
```

Parameters The function has the following parameters:

in_ppsmtpSink	Pointer to a pointer to a client structure associated with the response sink to free.

Returns

- If successful, the function returns NSMAIL_OK.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function frees the SMTPClient_t structure and its data members. When this function returns, the response sink is set to null. The user must free any opaque data.

Note Do not call the processResponses function after this function. §

See Also smtp_quit, smtp_free, smtpSink_initialize

[Top] [SMTP Functions] [SMTP Functions by Task]

---

smtpSink_initialize

Initializes and allocates the smtpSink_t structure.

Syntax

```
#include <smtp.h>
int smtpSink_initialize( smtpSink_t ** out_ppsmtpSink );
```

Parameters The function has the following parameters:

out_ppSMTPSink	Pointer to pointer to the response sink to initialize.

Returns

- If successful, the function returns NSMAIL_OK.

- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description For more information, see Creating a Response Sink.

Note Do not call the `processResponses` function after this function. §

See Also `smtp_connect, smtp_noop, smtpSink_free, smtp_initialize`

[Top] [SMTP Functions] [SMTP Functions by Task]

SMTP Structures

This section defines SMTP data structures, listed in alphabetical order.

`smtpClient_t` `smtpSink_t`

[Top] [SMTP Functions]

---

smtpClient_t

Represents the SMTP client.

Syntax

```
typedef struct smtpClient smtpClient_t;
```

Description The client uses this structure to communicate with an SMTP server. The developer uses this structure to perform SMTP API operations. All data contained within the client structure is used internally by the API.

For more information, see Creating a Client.

See Also `smtp_initialize, smtp_free`

[Top] [SMTP Structures]

smtpSink_t

Represents the SMTP notification sink.

Syntax

```
typedef struct smtpSink
{
   void * pOpaqueData;
   void (*bdat)( smtpSink_t * in_psmtpSink,
      int in_responseCode,
      const char * in_responseMessage );
   void (*connect)( smtpSink_t * in_psmtpSink,
      int in_responseCode,
      const char * in_responseMessage );
   void (*data)( smtpSink_t * in_psmtpSink,
      int in_responseCode,
      const char * in_responseMessage );
   void (*ehlo)( smtpSink_t * in_psmtpSink,
      int in_responseCode,
      const char * in_serverExtension );
   void (*ehloComplete)(smtpSink_t * in_psmtpSink );
   void (*expand)( smtpSink_t * in_psmtpSink,
      int in_responseCode,
      const char * in_emailAddress );
    void (*error)( SMTPSinkPtr_t in_psmtpSink,
      int in_responseCode,
      const char * in_errorMessage );
   void (*expand)( SMTPSinkPtr_t in_psmtpSink,
      int in_responseCode,
      const char * in_emailAddress );
   void (*expandComplete)(smtpSink_t * in_psmtpSink);
   void (*help)( smtpSink_t * in_psmtpSink,
      int in_responseCode,
      const char * in_helpMessage );
   void (*helpComplete)(smtpSink_t * in_psmtpSink );
   void (*mailFrom)( smtpSink_t * in_psmtpSink,
      int in_responseCode,
      const char * in_responseMessage );
   void (*noop)( smtpSink_t * in_psmtpSink,
      int in_responseCode,
      const char * in_responseMessage );
```

```
   void (*quit)( smtpSink_t * in_psmtpSink,
      int in_responseCode,
      const char * in_responseMessage );
   void (*rcptTo)( smtpSink_t * in_psmtpSink,
      int in_responseCode,
      const char * in_responseMessage );
   void (*reset)( smtpSink_t * in_psmtpSink,
      int in_responseCode,
      const char * in_responseMessage );
   void (*send)( smtpSink_t * in_psmtpSink,
      int in_responseCode,
      const char * in_responseMessage );
   void (*sendCommand)( smtpSink_t * in_psmtpSink,
      int in_responseCode,
      const char * in_responseMessage );
   void (*sendCommandComplete)(smtpSink_t *
      in_psmtpSink );
   void (*verify)( smtpSink_t * in_psmtpSink,
      int in_responseCode,
      const char * in_responseMessage );
}smtpSink_t;
```

Fields The structure has the following fields:

`void * pOpaqueData;`	User data.
`void (*bdat)( SMTPSinkPtr_t in_psmtpSink, int in_responseCode, const char * in_responseMessage );`	Notification for the response to the BDAT command. Parameters: the response sink to use, the SMTP response code, and descriptive response message.
`void (*connect)( SMTPSinkPtr_t in_psmtpSink, int in_responseCode, const char * in_responseMessage );`	Notification for the response to the connection to the server. Parameters: the response sink to use, the SMTP response code, and descriptive response message.
`void (*data)( SMTPSinkPtr_t in_psmtpSink, int in_responseCode, const char * in_responseMessage );`	Notification for the response to the DATA command. Parameters: the response sink to use, the SMTP response code, and descriptive response message.
`void (*ehlo)( SMTPSinkPtr_t in_psmtpSink, int in_responseCode, const char * in_serverExtension );`	Notification for the response to the EHLO command. Parameters: the response sink to use, the SMTP response code, and descriptive response message.
`void (*ehloComplete)( SMTPSinkPtr_t in_psmtpSink );`	Notification for the completion of the EHLO command. Parameter: response sink to use.

```
void (*error)(
   SMTPSinkPtr_t in_psmtpSink,
   int in_responseCode,
   const char * in_errorMessage );
```

Notification for an error. Parameters: the response sink to use, the SMTP response code, and descriptive response message.

```
void (*expand)(
   SMTPSinkPtr_t in_psmtpSink,
   int in_responseCode,
   const char * in_emailAddress );
```

Notification for the response to the EXPN command. Parameters: the response sink to use, the SMTP response code, and descriptive response message.

```
void (*expandComplete)(
   SMTPSinkPtr_t in_psmtpSink);
```

Notification for the completion of the EXPN command. Parameter: response sink for this client.

```
void (*help)(
   SMTPSinkPtr_t in_psmtpSink,
   int in_responseCode,
   const char * in_helpMessage );
```

Notification for the response to the HELP command. Parameters: the response sink to use, the SMTP response code, and descriptive response message.

```
void (*helpComplete)(
   SMTPSinkPtr_t in_psmtpSink );
```

Notification for the completion of the HELP command. Parameters: the response sink to use.

```
void (*mailFrom)(
   SMTPSinkPtr_t in_psmtpSink,
   int in_responseCode,
   const char * in_responseMessage );
```

Notification for the response to the MAIL FROM command. Parameters: the response sink to use, the SMTP response code, and descriptive response message.

```
void (*noop)(
   SMTPSinkPtr_t in_psmtpSink,
   int in_responseCode,
   const char * in_responseMessage );
```

Notification for the response to the NOOP command. Parameters: the response sink to use, the SMTP response code, and descriptive response message.

```
void (*quit)(
   SMTPSinkPtr_t in_psmtpSink,
   int in_responseCode,
   const char * in_responseMessage );
```

Notification for the response to the QUIT command. Parameters: the response sink to use, the SMTP response code, and descriptive response message.

```
void (*rcptTo)(
   SMTPSinkPtr_t in_psmtpSink,
   int in_responseCode,
   const char * in_responseMessage );
```

Notification for the response to the RCPT TO command. Parameters: the response sink to use, the SMTP response code, and descriptive response message.

```
void (*reset)(
   SMTPSinkPtr_t in_psmtpSink,
   int in_responseCode,
   const char * in_responseMessage );
```

Notification for the response to the RSET command. Parameters: the response sink to use, the SMTP response code, and descriptive response message.

`void (*send)( SMTPSinkPtr_t in_psmtpSink, int in_responseCode, const char * in_responseMessage );`	Notification for the response to data sent to the server. Parameters: the response sink to use, the SMTP response code, and descriptive response message.
`void (*sendCommand)( SMTPSinkPtr_t in_psmtpSink, int in_responseCode, const char * in_responseMessage );`	Notification for the response to `sendCommand` function. Parameters: the response sink to use, the SMTP response code, and descriptive response message.
`void (*sendCommandComplete) (SMTPSinkPtr_t in_psmtpSink );`	Notification for the completion of the extended command. Parameters: the response sink to use.
`void (*verify)( SMTPSinkPtr_t in_psmtpSink, int in_responseCode, const char * in_responseMessage );`	Notification for the response to the VRFY command. Parameters: the response sink to use, the SMTP response code, and descriptive response message.

Description The SMTP response sink structure is made up of function pointers and opaque data. You must define the functions yourself, and you must point these pointers to your functions if you want to receive the related server responses. For more information, see SDK Response Sinks for C and Creating a Response Sink.

The function pointers serve as callbacks for many SMTP functions. See SMTP Function Callback Mapping.

See Also `smtp_setResponseSink, smtpSink_initialize, smtpSink_free`

[Top] [SMTP Structures]

Chapter

10

MIME C API Reference

This chapter describes the functions and structures of the C language API for the MIME (Multipurpose Internet Mail Extension) protocol of the Messaging Access SDK.

- MIME Functions by Functional Group
- MIME Structures
- MIME Definitions

You'll find links to each function and structure in this introduction. Each reference entry gives the name of the function or structure, its header file, its syntax, and its parameters.

All C interface definitions are found in the `mime.h` and `mimeparser.h` files.

- The `mime.h` header file contains definitions common to the MIME encoder and the MIME parser APIs.
- The `mimeparser.h` header file contains definitions specific to the MIME parser API.

[Top]

MIME Functions by Functional Group

MIME functions are organized into functional groups. Click the group name to find the reference entries for the functions in the group. To find functions listed in alphabetical order by name, see MIME Functions by Name.

- Basic (Leaf) Body Part Functions
- Message Functions
- Message Part Functions

Composite Body Part Types

- Multipart Functions
- MIME Utility Functions
- Dynamic Parsing Functions
- Message Parsing Functions
- Data Sink Functions

[Top] [MIME Functions by Task]

MIME Functions by Name

This table lists MIME functions in alphabetical order by name. To find functions listed by functional group, see MIME Functions by Functional Group.

```
beginDynamicParse
dynamicParse
dynamicParseInputstream
endDynamicParse
getFileMIMEType
mime_basicPart_deleteData
mime_basicPart_free_all
mime_basicPart_getDataBuf
mime_basicPart_getDataStream
mime_basicPart_getSize
mime_basicPart_putByteStream
mime_basicPart_setDataBuf
mime_basicPart_setDataStream
mimeDataSink_free
mimeDataSink_new
mime_decodeBase64
mime_decodeHeaderString
mime_decodeQP
mimeDynamicParser_free
mimeDynamicParser_new
mime_encodeBase64
mime_encodeHeaderString
mime_encodeQP
mime_Header_free
mime_Header_new
mime_malloc
mime_memfree
mime_message_addBasicPart
mime_message_addMessagePart
mime_message_addMultiPart
mime_message_create
```

```
mime_message_deleteBody
mime_message_free_all
mime_message_getBody
mime_message_getContentSubType
mime_message_getContentType
mime_message_getContentTypeParams
mime_message_getHeader
mime_message_putByteStream
mime_messagePart_deleteMessage
mime_messagePart_free_all
mime_messagePart_fromMessage
mime_messagePart_getMessage
mime_messagePart_putByteStream
mime_messagePart_setMessage
mime_multiPart_addBasicPart
mime_multiPart_addFile
mime_multiPart_addMessagePart
mime_multiPart_addMultiPart
mime_multiPart_deletePart
mime_multiPart_free_all
mime_multiPart_getPart
mime_multiPart_getPartCount
mime_multiPart_putByteStream
parseEntireMessage
parseEntireMessageInputstream
```

[Top] [MIME Functions by Name] [MIME Functions by Task]

Basic (Leaf) Body Part Functions

The MIME BasicPart type includes all Basic MIME BodyPart types, including text, image, audio, video, and application.

This table lists Basic Part functions alphabetically by name, with descriptions. Click the function name to get information about it.

Function	Description
mime_basicPart_deleteData	Deletes the body data of the body part.
mime_basicPart_free_all	Frees the basic part, including all internal structures.
mime_basicPart_getDataBuf	Returns the decoded body data in a buffer.
mime_basicPart_getDataStream	Returns the decoded body data in an input stream.
mime_basicPart_getSize	Returns the size of the body data in bytes.
mime_basicPart_putByteStream	Writes the byte stream for this part with MIME headers and encoded body data.
mime_basicPart_setDataBuf	Sets the body data for this part from a buffer.
mime_basicPart_setDataStream	Sets the body data for this part from an input stream.

[Top] [MIME Functions by Functional Group]

mime_basicPart_deleteData

Deletes the body data of the body part.

Syntax

```
#include <mime.h>
int mime_basicPart_deleteData (
         mime_basicPart_t * in_pBasicPart);
```

Parameters The function has the following parameters:

`in_pBasicPart` Pointer to data to delete.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function deletes the bodyData from a part. After a message is built, you can use one of the MIME delete functions to delete the entire message, the body, or a message part as needed. For more information, see Deleting Parts of a Message.

See Also `mime_multiPart_deletePart, mime_message_deleteBody, mime_messagePart_deleteMessage`

[Top] [Basic (Leaf) Body Part Functions] [MIME Functions by Functional Group]

---

mime_basicPart_free_all

Frees the basic part, including all internal structures.

Syntax

```
#include <mime.h>
int mime_basicPart_free_all( mime_basicPart_t * in_pBasicPart);
```

Parameters The function has the following parameters:

`in_pBasicPart` Pointer to data to free.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function frees the basic part and its internal structures.

See Also `mime_multiPart_free_all, mime_message_free_all, mime_messagePart_free_all, mimeDynamicParser_free`

[Top] [Basic (Leaf) Body Part Functions] [MIME Functions by Functional Group]

---

mime_basicPart_getDataBuf

Returns the decoded body data in a buffer.

Syntax

```
#include <mime.h>
int mime_basicPart_getDataBuf (
            mime_basicPart_t * in_pBasicPart,
            unsigned int * out_pSize
            char ** out_ppDataBuf);
```

Parameters The function has the following parameters:

`in_pBasicPart`	Pointer to data to get.
`out_pSize`	Pointer to size of the basic part data.
`out_ppDataBuf`	Pointer to pointer to buffer that contains the data.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function returns the decoded body data in a buffer.

For more information, see Building the MIME Message.

See Also `mime_basicPart_getDataStream, mime_basicPart_setDataBuf, mime_basicPart_setDataStream`

[Top] [Basic (Leaf) Body Part Functions] [MIME Functions by Functional Group]

---

mime_basicPart_getDataStream

Returns the decoded body data as an input stream.

Syntax

```
#include <mime.h>
int mime_basicPart_getDataStream (
            mime_basicPart_t * in_ppBasicPart,
            char * in_pFileName,
            nsmail_inputstream_t ** out_ppDataStream);
```

Parameters The function has the following parameters:

`in_ppBasicPart`	Pointer to data to retrieve.
`in_pFileName`	Pointer to the name of a file to retrieve. If `null`, returns a memory based input-stream.
`out_ppDataStream`	Pointer to pointer to output data stream.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function returns the file named in the `in_pFileName` parameter. If this parameter is `null`, the function returns a memory-based input-stream.

For more information, see Building the MIME Message.

See Also
```
mime_basicPart_getDataBuf,
mime_basicPart_setDataBuf,
mime_basicPart_setDataStream
```

[Top] [Basic (Leaf) Body Part Functions] [MIME Functions by Functional Group]

---

mime_basicPart_getSize

Returns the size of the body data in bytes.

Syntax
```
#include <mime.h>
int mime_basicPart_getSize (
            mime_basicPart_t * in_pBasicPart,
            unsigned int * out_pSize);
```

Parameters The function has the following parameters:

`in_pBasicPart`	Pointer to data to get size for.
`out_pSize`	Pointer to size of body data to get.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description For more information, see Building the MIME Message.

See Also `mime_basicPart_getDataBuf, mime_basicPart_getDataStream`

[Top] [Basic (Leaf) Body Part Functions] [MIME Functions by Functional Group]

---

mime_basicPart_putByteStream

Writes the byte stream for this part with MIME headers and encoded body data.

Syntax

```
#include <mime.h>
int mime_basicPart_putByteStream (
            mime_basicPart_t * in_pBasicPart,
            mime_outputstream_t * in_pOutput_stream);
```

Parameters The function has the following parameters:

`in_pBasicPart`	Pointer to data to write.
`in_pOutput_stream`	Pointer to MIME output stream.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function encodes the specified basic part with MIME headers and encoded body data. This places the part in MIME canonical format, so that it can be transported using SMTP or any other transport method.

For more information, see Encoding the Message.

See Also

```
mime_messagePart_putByteStream,
mime_message_putByteStream,
mime_multiPart_putByteStream
```

[Top] [Basic (Leaf) Body Part Functions] [MIME Functions by Functional Group]

---

mime_basicPart_setDataBuf

Sets the body data for this part from a buffer.

Syntax

```
#include <mime.h>
int mime_basicPart_setDataBuf (
         mime_basicPart_t * in_pBasicPart,
         unsigned int in_size,
         const char * in_pDataBuf)
         boolean in_fCopyData);
```

Parameters The function has the following parameters:

`in_pBasicPart`	Pointer to body data.
`in_size`	Size of body data.
`in_pDataBuf`	Pointer to buffer that contains body data.
`in_fCopyData`	Whether the message copies the data or keeps a reference to the data buffer. • `true`: Keep a reference to the buffer. • `false`: Copy the data.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description You can add data only to a new, already created `mime_basicPart` with empty body data. For more information, see Adding Content to the Message.

See Also `mime_basicPart_getDataBuf, mime_basicPart_getDataStream, mime_malloc, mime_memfree`

[Top] [Basic (Leaf) Body Part Functions] [MIME Functions by Functional Group]

---

mime_basicPart_setDataStream

Sets the body data for this part from an input stream.

Syntax

```
#include <mime.h>
```

```
int mime_basicPart_setDataStream (
        mime_basicPart_t * in_pBasicPart,
        mime_inputstream_t * in_ptheDataStream
        boolean in_fCopyData);
```

Parameters The function has the following parameters:

`in_pBasicPart`	Pointer to the body data to set.
`in_ptheDataStream`	Pointer to the MIME input stream.
`in_fCopyData`	Whether the message copies the data or keeps a reference to the data buffer. • `true`: Keep a reference to the buffer. • `false`: Copy the data.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description You can add data only to new, already created `mime_basicPart` with empty body data. For more information, see Adding Content to the Message.

See Also `mime_basicPart_getDataBuf, mime_basicPart_getDataStream, mime_basicPart_setDataBuf`

[Top] [Basic (Leaf) Body Part Functions] [MIME Functions by Functional Group]

Message Functions

The message content type is part of the Composite Body Part Types Group. Message functions create, add to, or perform other operations on the message as a whole.

This table lists Message functions alphabetically by name, with descriptions. Click the function name to get information about it.

Function	Description
mime_message_addBasicPart	Adds a basic part as the body of the message.
mime_message_addMessagePart	Adds the message part as the body of the message.
mime_message_addMultiPart	Adds the multipart as the body of the message.
mime_message_create	Creates a message given text-data and a file to attach.
mime_message_deleteBody	Deletes the body part that makes up the body of this message.
mime_message_free_all	Frees the message, including all internal structures.
mime_message_getBody	Returns the body part that makes up the body of the message.
mime_message_getContentSubType	Returns the content subtype of this message.
mime_message_getContentType	Returns the content type of the body of this message.
mime_message_getContentTypeParams	Returns the content type parameters of this message.
mime_message_getHeader	Returns the header value, given the header name.
mime_message_putByteStream	Writes the encoded byte stream for this part to an output stream.

[Top] [MIME Functions by Functional Group]

mime_message_addBasicPart

Adds a basic part as the body of the message.

Syntax

```
#include <mime.h>
int mime_message_addBasicPart (
            mime_message_t * in_pMessage,
            mime_basicPart_t * in_pBasicPart,
            BOOLEAN in_fClone);
```

Parameters The function has the following parameters:

`in_pMessage`	Pointer to message.
`in_pBasicPart`	Pointer to basic part to add.
`in_fClone`	Whether the message keeps a reference to the basic part or a cloned copy. Values: • `true`: Keep a reference to the basic part. • `false`: Keep a cloned copy.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function sets the content type according to the message type. If the message body is already present, an error occurs.

For more information, see Building the MIME Message.

See Also `mime_multiPart_addBasicPart, mime_multiPart_addMessagePart, mime_multiPart_addMultiPart, mime_multiPart_deletePart, mime_message_addBasicPart, mime_message_addMultiPart`

[Top] [Message Functions] [MIME Functions by Functional Group]

---

mime_message_addMessagePart

Adds the message part as the body of the message.

Syntax

```
#include <mime.h>
int mime_message_addMessagePart (
                mime_message_t * in_pMessage,
                mime_messagePart_t * in_pMessagePart,
                BOOLEAN in_fClone);
```

Parameters The function has the following parameters:

`in_pMessage`	Pointer to message.
`in_pMessagePart`	Pointer to message part to add.
`in_fClone`	Whether the message keeps a reference to the basic part structure or a cloned copy. Values: • `true`: Keep a reference to the basic part. • `false`: Keep a cloned copy.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function adds the message part and sets the message content type to message part to match the added part. If the message body is already present, an error occurs. For more information, see Building the MIME Message.

See Also `mime_multiPart_addBasicPart, mime_multiPart_addMessagePart, mime_multiPart_addMultiPart, mime_multiPart_deletePart, mime_message_addBasicPart, mime_message_addMultiPart`

[Top] [Message Functions] [MIME Functions by Functional Group]

---

mime_message_addMultiPart

Adds a multipart as the body of the message.

Syntax

```
#include <mime.h>
int mime_message_addMultiPart (
                mime_message_t * in_pMessage,
                mime_multiPart_t * in_pMultiPart,
                BOOLEAN in_fClone);
```

Parameters The function has the following parameters:

`in_pMessage`	Pointer to message.
`in_pMessagePart`	Pointer to multipart to add.
`in_fClone`	Whether the message keeps a reference to the basic part structure or a cloned copy. Values: • `true`: Keep a reference to the basic part. • `false`: Keep a cloned copy.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description A multipart type is made up of one or more body parts. This function adds a multipart and sets the message content type to multipart to match the type of the added part. If the message body is already present, an error occurs. For more information, see Building the MIME Message.

See Also `mime_multiPart_addBasicPart, mime_multiPart_addMessagePart, mime_multiPart_deletePart, mime_message_addBasicPart, mime_message_addMessagePart`

[Top] [Message Functions] [MIME Functions by Functional Group]

---

mime_message_create

Creates a message given text-data and a file to attach.

Syntax

```
#include <mime.h>
int mime_message_create (char * in_pText,
             char * in_pFileFullName,
             mime_encoding_type in_perf_file_encoding,
             mime_message_t ** out_ppMessage);
```

Parameters The function has the following parameters:

`in_pText`	Pointer to message text data. Required if `in_pFileFullName` is `null`.
`in_pFileFullName`	Pointer to fully-qualified file-name of file to attach. Required if `in_pText` is `null`.
`in_perf_file_encoding`	Encoding type. For values, see MIME Encoding Types. • > 0: function attempts to use the encoding for the file-attachment. • If value is not valid for the file-type, overrides with the correct value.
`out_ppMessage`	Pointer to pointer to message to create.

Returns
- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function creates a MIME message given either a file attachment or text data, or both. Either a fully-qualified file-name for the `in_pFileFullName` parameter or the text data for the `in_pText` parameter is required. MIME creates a message with the parameter it gets, but cannot if both parameters are `null`; in this case it returns an error.

Note Values for the `in_pText` and `in_pFileFullName` parameters can be found upon return from this call. See `mime_memfree`. §

For more information, see Building the MIME Message.

See Also `mime_multiPart_addBasicPart, mime_multiPart_addMessagePart, mime_multiPart_deletePart, mime_message_addBasicPart, mime_message_addMessagePart, mime_malloc, mime_memfree`

[Top] [Message Functions] [MIME Functions by Functional Group]

---

mime_message_deleteBody

Deletes the body part that makes up the body of this message.

Syntax

```
#include <mime.h>
int mime_message_deleteBody (
              mime_message_t * in_pMessage);
```

Parameters The function has the following parameters:

`in_pMessage`	Pointer to message that contains the body part to delete.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function deletes a body part that is a message body. After a message is built, you can use one of the MIME delete functions to delete the entire message, the body, or a message part as needed. For more information, see Deleting Parts of a Message.

See Also `mime_basicPart_deleteData, mime_multiPart_deletePart, mime_messagePart_deleteMessage`

[Top] [Message Functions] [MIME Functions by Functional Group]

---

mime_message_free_all

Frees the message including all internal structures.

Syntax

```
#include <mime.h>
int mime_message_free_all ( mime_message_t * in_pMessage);
```

Parameters The function has the following parameters:

`in_pMessage`	Pointer to the message to free.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function frees the message, including all internal data structures.

See Also `mime_basicPart_free_all, mime_multiPart_free_all,`

mimeDynamicParser_free, mime_messagePart_free_all

[Top] [Message Functions] [MIME Functions by Functional Group]

---

mime_message_getBody

Returns the body part that makes up the body of the message.

Syntax

```
#include <mime.h>
int mime_message_getBody ( mime_message_t * pMessage,
                     mime_content_type * in_pContentType,
                     void ** out_ppTheBodyPart;
```

Parameters The function has the following parameters:

in_pMessage	Pointer to the message.
in_pContentType	Pointer to the MIME Content type. For values, see MIME Content Types.
out_ppTheBodyPart	Pointer to pointer to the body part. The client can cast this based on the value of in_pContentType.

Returns

- If successful, the function returns MIME_OK.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

See Also mime_message_getContentType

[Top] [Message Functions] [MIME Functions by Functional Group]

---

mime_message_getContentSubType

Returns the content subtype of this message.

Syntax

```
#include <mime.h>
int mime_message_getContentSubType (
```

```
                                mime_message_t * in_pMessage,
                                char ** out_ppSubType);
```

Parameters The function has the following parameters:

`in_pMessage`	Pointer to message.
`out_ppSubType`	Pointer to pointer to the MIME subtype of the message.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function returns the MIME subtype of this message. For more information, see MIME Content Types.

See Also `mime_message_getContentType, mime_message_getContentTypeParams`

[Top] [Message Functions] [MIME Functions by Functional Group]

---

mime_message_getContentType

Returns the content type of this message.

Syntax

```
#include <mime.h>
int mime_message_getContentType (
                mime_message_t * in_pMessage,
                mime_content_type * out_content_type);
```

Parameters The function has the following parameters:

`in_pMessage`	Pointer to the message.
`out_content_type`	Pointer to the MIME content type. For values, see MIME Content Types.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description The content type can be text, image, audio, video, or application. For more information, see MIME Content Types.

See Also `mime_message_getContentTypeParams`

[Top] [Message Functions] [MIME Functions by Functional Group]

---

mime_message_getContentTypeParams

Returns the `content_type` parameters of this message.

Syntax

```
#include <mime.h>
int mime_message_getContentTypeParams (
                    mime_message_t * in_pMessage,
                    char ** out_ppParams);
```

Parameters The function has the following parameters:

`in_pMessage`	Pointer to message.
`out_pParams`	Pointer to pointer to the content type parameters of the message.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

See Also `mime_message_getContentType`

[Top] [Message Functions] [MIME Functions by Functional Group]

---

mime_message_getHeader

Returns the header value, given the header name.

Syntax

```
#include <mime.h>
```

```
int mime_message_getHeader (
             mime_message_t * in_pMessage,
             char * in_pName,
             char ** out_ppValue);
```

Parameters The function has the following parameters:

`in_pMessage`	Pointer to message.
`in_pName`	Pointer to the header name.
`out_ppValue`	Pointer to pointer to the header value.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function returns the value of the header, given the header name, it is in the message).

See Also `mime_message_getContentType`

[Top] [Message Functions] [MIME Functions by Functional Group]

---

mime_message_putByteStream

Writes the encoded byte stream for this part to an output stream.

Syntax

```
#include <mime.h>
int mime_message_putByteStream (
            mime_message_t * in_pMessage,
            nsmail_outputstream_t * in_pOutput_stream);
```

Parameters The function has the following parameters:

`in_pMessage`	Pointer to message that contains the body part.
`in_pOutput_stream`	Pointer to the MIME output stream to write to.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function encodes the specified part with MIME headers and encoded body data. This places the part in MIME canonical format, so that it can be transported using SMTP or any other transport method.

For more information, see Encoding the Message.

See Also `mime_multiPart_putByteStream, mime_messagePart_putByteStream`

[Top] [MIME Functions by Functional Group] [MIME Functions by Functional Group] [MIME Functions by Task]

Message Part Functions

The Message Part body part represents a message that becomes the content of a another message, for example, when forwarded.

This table lists Message Part functions alphabetically by name, with descriptions. Click the function name to get information about it.

Function	Description
`mime_messagePart_deleteMessage`	Deletes the MIME message that makes up the body of this part.
`mime_messagePart_free_all`	Frees the message part, including all internal structures.
`mime_messagePart_fromMessage`	Creates a message part from a message structure.
`mime_messagePart_getMessage`	Retrieves the message from the message part.
`mime_messagePart_putByteStream`	Writes the encoded byte stream for this part with MIME headers and encoded body data.
`mime_messagePart_setMessage`	Sets a message as the body of this message part.

[Top] [MIME Functions by Functional Group]

mime_messagePart_deleteMessage

Deletes the MIME message that makes up the body of this part.

Syntax

```
#include <mime.h>
int mime_messagePart_deleteMessage (
            mime_messagePart_t * in_pMessagePart);
```

Parameters The function has the following parameters:

`in_pMessagePart` Pointer to message part.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function deletes a forwarded MIME message from a part. After a message is built, you can use one of the MIME delete functions to delete the entire message, the body, or a message part as needed. For more information, see Deleting Parts of a Message.

See Also `mime_basicPart_deleteData, mime_multiPart_deletePart, mime_message_deleteBody`

[Top] [Message Part Functions] [MIME Functions by Functional Group]

---

mime_messagePart_free_all

Frees the message part, including all internal structures.

Syntax

```
#include <mime.h>
int mime_messagePart_free_all (
         mime_messagePart_t * in_pMessagePart);
```

Parameters The function has the following parameter:

`in_pMessagePart` Pointer to message part to free.

Returns

- If successful, the function returns `MIME_OK`.

- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function frees the message part and its internal structures.

See Also `mime_basicPart_free_all, mime_multiPart_free_all, mime_message_free_all, mimeDynamicParser_free`

[Top] [Message Part Functions] [MIME Functions by Functional Group]

---

mime_messagePart_fromMessage

Creates a message part from a message structure.

Syntax

```
#include <mime.h>
int mime_messagePart_fromMessage (
            mime_message_t * in_pMessage,
            mime_messagePart_t ** out_ppMessagePart);
```

Parameters The function has the following parameters:

`in_pMessage`	Pointer to message structure to use to create part.
`out_ppMessagePart`	Pointer to pointer to message part to create.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description For more information, see Adding Content to the Message.

See Also `mime_messagePart_getMessage, mime_message_create`

[Top] [Message Part Functions] [MIME Functions by Functional Group]

---

mime_messagePart_getMessage

Retrieves the message from the message part.

Syntax

```
#include <mime.h>
int mime_messagePart_getMessage (
            mime_messagePart_t * in_pMessagePart,
            mime_message_t ** out_ppMessage);
```

Parameters The function has the following parameters:

`in_pMessage`	Pointer to message structure.
`out_ppMessagePart`	Pointer to pointer to message part.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

See Also `mime_messagePart_fromMessage, mime_messagePart_setMessage`

[Top] [Message Part Functions] [MIME Functions by Functional Group]

---

mime_messagePart_putByteStream

Writes the byte stream for this part to an output stream.

Syntax

```
#include <mime.h>
int mime_messagePart_putByteStream (
            mime_messagePart_t * in_pMessagePart,
            mime_outputstream_t * in_pOutput_stream);
```

Parameters The function has the following parameters:

`in_pMessagePart`	Pointer to message part.
`in_pOutput_stream`	Pointer to MIME output stream.

Returns

- If successful, the function returns `MIME_OK`.

- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description For more information, see Encoding the Message.

See Also `mime_multiPart_putByteStream, mime_message_putByteStream`

[Top] [Message Part Functions] [MIME Functions by Functional Group]

---

mime_messagePart_setMessage

Sets the parsed message as the body of this message part.

Syntax

```
#include <mime.h>
int mime_messagePart_setMessage (
            mime_messagePart_t * in_pMessagePart,
            mime_message_t * in_pMessage);
```

Parameters The function has the following parameters:

`in_pMessagePart`	Pointer to message part.
`in_pMessage`	Pointer to message.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description If you try to set a message that has already been set, you get an error message. For more information, see Building the MIME Message.

See Also `mime_messagePart_getMessage`

[Top] [Message Part Functions] [MIME Functions by Functional Group]

Multipart Functions

The multipart content type is part of the Composite Body Part Types Group. It describes a message that is made up of one or more sub-body parts. The multipart type has several subtypes that describe how the sub-parts relate to each other, including mixed, alternative, digest, and parallel.

This table lists Multipart functions alphabetically by name, with descriptions. Click the function name to get information about it.

Function	Description
mime_multiPart_addBasicPart	Adds a basic part to the multipart.
mime_multiPart_addFile	Adds a file as basic part to the multipart.
mime_multiPart_addMessagePart	Adds a message part to the multipart.
mime_multiPart_addMultiPart	Adds a message part to the multipart.
mime_multiPart_deletePart	Deletes the body part at the specified index from this multipart.
mime_multiPart_free_all	Frees the multipart, including all internal structures.
mime_multiPart_getPart	Returns the body part at the specified index.
mime_multiPart_getPartCount	Returns the count of the body parts in this multipart.
mime_multiPart_putByteStream	Writes the byte stream for this part with MIME headers and encoded body data.

[Top] [MIME Functions by Functional Group]

mime_multiPart_addBasicPart

Adds a basic part to the multipart.

Syntax

```
#include <mime.h>
int mime_multiPart_addBasicPart (
            mime_multiPart_t * in_pMultiPart,
```

```
                mime_basicPart_t * in_pBasicPart,
                boolean in_fClone,
                int * out_pIndex_assigned);
```

Parameters The function has the following parameters:

`in_pMultiPart`	Pointer to multipart to add basic part to.
`in_pBasicPart`	Pointer to basic part to add.
`in_fClone`	Whether the message keeps a reference to the basic part structure or a cloned copy. Values: • `true`: Keep a reference to the basic part. • `false`: Keep a cloned copy.
`out_pIndex_assigned`	Pointer to index where the basic part should be added.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function adds an existing basic part to a multipart at the specified index. The `in_fClone` parameter specifies whether the function returns the body part or a cloned copy. For more information, see Adding a Basic Part to a Multipart.

See Also `mime_multiPart_addMessagePart, mime_multiPart_addMultiPart, mime_multiPart_deletePart, mime_message_addBasicPart, mime_message_addMessagePart, mime_message_addMultiPart`

[Top] [Multipart Functions] [MIME Functions by Functional Group]

---

mime_multiPart_addFile

Adds a file as basic part to the multipart.

Syntax

```
#include <mime.h>
int mime_multiPart_addFile (
         mime_multiPart_t * in_pMultiPart,
         char * in_pFileFullName,
         mime_encoding_type in_perf_file_encoding,
         int * out_index_assigned);
```

Parameters The function has the following parameters:

`in_pMultiPart`	Pointer to multipart to add file to.
`in_pFileFullName`	Pointer to fully-qualified filename of existing file to attach.
`in_perf_file_encoding`	Encoding type. For values, see MIME Encoding Types. • > 0: function attempts to use the encoding for the file-attachment. • If value is not valid for the file-type, overrides with the correct value.
`out_index_assigned`	Pointer to index.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description The file added by this function must already exist and you must identify it with a fully-qualified file-name including PATH. If either the `in_pText` or `in_pFileFullName` parameter is `null`, the function creates a MIME message with the non-`null` parameter. If both parameters are `null`, an error is returned.

For more information, see Building the MIME Message.

See Also `mime_multiPart_addMessagePart, mime_multiPart_addMultiPart, mime_multiPart_deletePart, mime_message_addBasicPart, mime_message_addMessagePart, mime_message_addMultiPart`

[Top] [Multipart Functions] [MIME Functions by Functional Group]

---

mime_multiPart_addMessagePart

Adds a message part to the multipart.

Syntax

```
#include <mime.h>
int mime_multiPart_addMessagePart (
                  mime_multiPart_t * in_pMultiPart,
                  mime_messagePart_t * in_pMessagePart,
                  BOOLEAN in_fClone,
                  int * out_Index_assigned);
```

Parameters The function has the following parameters:

`in_pMultiPart`	Pointer to multipart to add message part to.
`in_pMessagePart`	Pointer to message part to add.
`in_fClone`	Whether the message keeps a reference to the basic part structure or a cloned copy. Values: • `true`: Keep a reference to the basic part. • `false`: Keep a cloned copy.
`out_Index_assigned`	Pointer to index.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description For more information, see Building the MIME Message.

See Also `mime_multiPart_addBasicPart, mime_multiPart_addMultiPart, mime_multiPart_deletePart, mime_message_addBasicPart, mime_message_addMessagePart, mime_message_addMultiPart`

[Top] [Multipart Functions] [MIME Functions by Functional Group]

---

mime_multiPart_addMultiPart

Adds a multipart to the multipart.

Syntax

```
#include <mime.h>
int mime_multiPart_addMultiPart (
         mime_multiPart_t * in_pMultiPart,
         mime_multiPart_t * in_pMultiPartToAdd,
         boolean in_fClone,
         int * out_Index_assigned);
```

Parameters The function has the following parameters:

`in_pMultiPart`	Pointer to multipart to which to add the message part.
`in_pMultiPartToAdd`	Pointer to multipart to add.
`in_fClone`	Whether the message keeps a reference to the basic part structure or a cloned copy. Values: • `true`: Keep a reference to the basic part. • `false`: Keep a cloned copy.
`out_Index_assigned`	Pointer to index.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description For more information, see Building the MIME Message.

See Also `mime_multiPart_addBasicPart, mime_multiPart_addMessagePart, mime_multiPart_deletePart, mime_message_addBasicPart, mime_message_addMessagePart, mime_message_addMultiPart`

[Top] [Multipart Functions] [MIME Functions by Functional Group]

---

mime_multiPart_deletePart

Deletes the body part at the requested index from this multipart.

Syntax

```
#include <mime.h>
int mime_multiPart_deletePart (
                  mime_multiPart_t * in_pMultiPart,
                  int in_index);
```

Parameters The function has the following parameters:

`in_pMultiPart`	Pointer to multipart from which to delete the message part.
`in_index`	Index.

Returns

- If successful, the function returns `MIME_OK`.

- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function deletes the body part at the specified index. After a message is built, you can use one of the MIME delete functions to delete the entire message, the body, or a message part as needed. For more information, see Deleting Parts of a Message.

See Also `mime_basicPart_deleteData`, `mime_messagePart_deleteMessage`, `mime_message_deleteBody`

[Top] [Multipart Functions] [MIME Functions by Functional Group]

---

mime_multiPart_free_all

Frees the multipart including all internal structures.

Syntax

```
#include <mime.h>
int mime_multiPart_free_all (
         mime_multiPart_t * in_pMultiPart);
```

Parameters The function has the following parameters:

`in_pMultiPart`	Pointer to multipart to free.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function frees the multipart and its internal structures.

See Also `mime_basicPart_free_all`, `mime_message_free_all`, `mime_messagePart_free_all`, `mimeDynamicParser_free`

[Top] [Multipart Functions] [MIME Functions by Functional Group]

---

mime_multiPart_getPart

Returns the body part at the specified index.

Syntax

```
#include <mime.h>
int mime_multiPart_getPart (
            mime_multiPart_t * in_pMultiPart,
            int in_index,
            mime_content_type * in_pContentType,
            void ** out_ppTheBodyPart );
```

Parameters The function has the following parameters:

`in_pMultiPart`	Pointer to multipart.
`in_index`	Index target for body part.
`in_pContentType`	Pointer to MIME Content type. For values, see MIME Content Types.
`out_ppTheBodyPart`	Pointer to pointer to body part. Client can cast this based on the type in the `in_pContentType` parameter.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function returns the body part, specified by its index in the `in_index` parameter.

See Also `mime_multiPart_getPartCount`

[Top] [Multipart Functions] [MIME Functions by Functional Group]

---

mime_multiPart_getPartCount

Returns the count of the body parts in this multipart.

Syntax

```
#include <mime.h>
int mime_multiPart_getPartCount (
            mime_multiPart_t * in_pMultiPart,
```

```
                int * count);
```

Parameters The function has the following parameters:

`in_pMultiPart`	Pointer to multipart for which to count parts.
`count`	Pointer to number of parts.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

See Also `mime_multiPart_getPart`

[Top] [Multipart Functions] [MIME Functions by Functional Group]

---

mime_multiPart_putByteStream

Writes out the encoded byte stream for this multipart with MIME headers and encoded body data.

Syntax

```
#include <mime.h>
int mime_multiPart_putByteStream (
              mime_multiPart_t * in_pMultiPart,
              nsmail_outputstream_t * in_pOutput_stream);
```

Parameters The function has the following parameters:

`in_pMultiPart`	Pointer to multipart.
`out_pOutput_stream`	Pointer to MIME output stream.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function encodes the specified multipart with MIME headers and encoded body data. This places the part in MIME canonical format, so that it can be transported using SMTP or any other transport method.

For more information, see Encoding the Message.

See Also `mime_messagePart_putByteStream, mime_message_putByteStream, mime_basicPart_putByteStream`

[Top] [Multipart Functions] [MIME Functions by Functional Group]

MIME Utility Functions

MIME utility functions perform specialized encoding and decoding. In addition, two convenience functions create and destroy headers.

This table lists MIME Utility functions alphabetically by name, with descriptions. Click the function name to get information about it.

Function	Description
`getFileMIMEType`	Returns the MIME type information for the specified file, based on file extension.
`mime_decodeBase64`	Decodes Base64-encoded data and writes it to an output stream.
`mime_decodeHeaderString`	Decodes and returns a header string.
`mime_decodeQP`	Decodes Quoted Printable-encoded data and writes it to an output stream.
`mime_encodeBase64`	Encodes the data from an input stream and writes it to an output stream, using Base64 encoding.
`mime_encodeHeaderString`	Encodes a header string.
`mime_encodeQP`	Encodes data using QP encoding.
`mime_Header_free`	Frees a MIME header.
`mime_Header_new`	Builds and returns a MIME header, given header name and value.
`mime_malloc`	Allocates memory for use with the MIME API.
`mime_memfree`	Frees memory allocated by the SDK.

[Top] [MIME Functions by Functional Group]

mime_decodeBase64

Decodes Base64-encoded data and writes it to the output stream.

Syntax

```
#include <mime.h>
int mime_decodeBase64 (
            nsmail_inputstream_t * in_pInput_stream,
            nsmail_outputstream_t * in_pOutput_stream);
```

Parameters The function has the following parameters:

`in_pInput_stream`	Pointer to the MIME input stream source of data to encode.
`out_pOutput_stream`	Pointer to the MIME output stream target for encoded data.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function decodes Base-64-encoded data from the `input_stream` and writes it to the `output_stream`. Base64 is the default encoding for non-Text content types.

See Also `mime_decodeHeaderString, mime_decodeQP, mime_encodeBase64`

[Top] [MIME Utility Functions] [MIME Functions by Functional Group]

mime_decodeHeaderString

Decodes and returns a header string.

Syntax

```
#include <mime.h>
int mime_decodeHeaderString (char * inString,
                        char ** out_ppString);
```

Parameters The function has the following parameters:

`inString`	Pointer to header string that was encoded according to RFC 2047 rules.
`out_ppString`	Pointer to pointer to decoded output.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description If the input string is not encoded according to RFC 2047 rules, this function returns the same input string. Otherwise, it decodes and returns the decoded string.

For more information, see Parsing the Entire Message.

See Also `mime_decodeBase64, mime_decodeQP, mime_encodeHeaderString`

[Top] [MIME Utility Functions] [MIME Functions by Functional Group]

---

mime_decodeQP

Decodes Quoted Printable-encoded data and writes it to the output stream.

Syntax

```
#include <mime.h>
int mime_decodeQP (mime_inputstream_t * in_pInput_stream,
                   mime_outputstream_t * out_pOutput_stream
```

Parameters The function has the following parameters:

`in_pInput_stream`	Pointer to the MIME input stream source of data to encode.
`out_pOutput_stream`	Pointer to the MIME output stream target for encoded data.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function decodes QuotedPrintable-encoded data from the `input_stream` and writes it to the `output_stream`. For more information, see Parsing the Entire Message

See Also `mime_decodeBase64, mime_decodeHeaderString, mime_encodeQP`

[Top] [MIME Utility Functions] [MIME Functions by Functional Group]

---

mime_encodeBase64

Encodes the data from an input stream and writes it to an output stream, using Base64 encoding.

Syntax

```
#include <mime.h>
int mime_encodeBase64 (
         mime_inputstream_t * in_pInput_stream,
         mime_outputstream_t * out_pOutput_stream);
```

Parameters The function has the following parameters:

`in_pInput_stream`	Pointer to the MIME input stream source of data to encode.
`out_pOutput_stream`	Pointer to the MIME output stream target for encoded data.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description For more information, see Encoding and Decoding Utilities.

See Also `mime_encodeHeaderString, mime_encodeQP, mime_decodeBase64`

[Top] [MIME Utility Functions] [MIME Functions by Functional Group]

---

mime_encodeHeaderString

Encodes a header string.

Syntax

```
#include <mime.h>
int mime_encodeHeaderString (char * inString,
             char * in_charset,
             char in_encoding,
             char ** out_ppOutString);
```

Parameters The function has the following parameters:

`inString`	Pointer to the header string to encode. Header must conformity RFC 2047 requirements.
`in_charset`	Pointer to the character set that contains the input string.
`in_encoding`	Encoding type. Values: 'Q' or 'B'
`out_ppOutString`	Pointer to pointer to the MIME output stream.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function encodes a header string as defined in RFC 2047. The encoded string can be used as the value of unstructured headers or in the comments section of structured headers. For a complete list, see Encoding and Decoding Headers.

See Also `mime_encodeBase64, mime_encodeQP, mime_decodeHeaderString`

[Top] [MIME Utility Functions] [MIME Functions by Functional Group]

---

mime_encodeQP

Encodes the data from an `input_stream` and writes to an `output_stream`, using QP encoding.

Syntax

```
#include <mime.h>
int mime_encodeQP (
```

```
mime_inputstream_t * in_pInput_stream,
mime_outputstream_t * out_pOutput_stream);
```

Parameters The function has the following parameters:

`in_pInput_stream`	Pointer to the MIME input stream source of data to encode.
`out_pOutput_stream`	Pointer to the MIME output stream target for encoded data.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description For more information, see Encoding and Decoding Utilities.

See Also `mime_encodeBase64, mime_encodeHeaderString, mime_encodeQP`

[Top] [MIME Utility Functions] [MIME Functions by Functional Group]

---

mime_Header_free

Frees a MIME header.

Syntax

```
#include <mime.h>
int mime_header_free (mime_header_t * in_pHdr);
```

Parameters The function has the following parameters:

`in_pHdr`	Pointer to the header string to free.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function frees a `mime_header_t` structure.

See Also `mime_Header_new, mime_header_t, mime_basicPart_free_all, mime_multiPart_free_all, mime_message_free_all, mime_messagePart_free_all`

[Top] [MIME Utility Functions] [MIME Functions by Functional Group]

mime_Header_new

Builds and returns a MIME header, given header name and value.

Syntax

```
#include <mime.h>
mime_header_t * mime_header_new (char * in_pName,
                                 char * in_pValue);
```

Parameters The function has the following parameters:

`in_pName`	Pointer to the name portion of the header.
`pValue`	Pointer to the value portion of the header.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function can create a `mime_header_t` structure with either or both of the name and value parameters. If both are `null`, the function returns `null`. For more information, see Adding Message Headers.

See Also `mime_header_t`, `mime_Header_free`

[Top] [MIME Utility Functions] [MIME Functions by Functional Group]

getFileMIMEType

Returns the MIME type information for the specified file, based on file extension.

Syntax

```
#include <mime.h>
int getFileMIMEType (char * in_pFilename_ext,
                     file_mime_type * in_pFileMIMEType);
```

Parameters The function has the following parameters:

`in_pFilename_ext`	Pointer to the file extension. If `null`, returns application/octet-stream.
`in_pFileMIMEType`	Pointer to the MIME type of the file. Default: application/octet-stream.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description Based on file extension, this method attempts to determine the file MIME type. By default, it returns the application/octet-stream.

This helper method is useful for attaching files and setting their MIME-types automatically.

See Also `mime_encodeBase64, mime_encodeHeaderString, mime_encodeQP`

[Top] [MIME Utility Functions] [MIME Functions by Functional Group]

---

mime_malloc

Allocates memory for use with the MIME API.

Syntax

```
#include <mime.h>
void * mime_malloc (unsigned int in_size);
```

Parameters The function has the following parameter:

`in_size`	Size of memory to allocate.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function allocates memory of the specified size. It is preferable to use this function instead of `malloc` for memory you are allocating for use with the MIME API. By using `mime_malloc`, you avoid heap space conflicts that can occur when the mail application links with two or more libraries that support `malloc` and `free`.

Internally, this function simply invokes malloc, so its semantics and usage are identical to those of `malloc`.

See Also `mime_memfree`

[Top] [MIME Utility Functions] [MIME Functions by Functional Group]

---

mime_memfree

Frees memory allocated by the SDK.

Syntax

```
#include <mime.h>
void mime_memfree (void * in_pMem);
```

Parameters The function has the following parameter:

`in_pMem`	Memory to free.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function frees any memory allocated by the SDK. This includes data buffers returned by `mime_basicPart_getDataBuf` and memory allocated using `mime_malloc`.

Internally, this function simply invokes `free`, so its semantics and usage are identical to those of `free`.

See Also `mime_malloc, mime_basicPart_getDataBuf`

[Top] [MIME Utility Functions] [MIME Functions by Functional Group]

Dynamic Parsing Functions

These functions provide the capabilities of the dynamic parser. For more information, see Dynamic Parsing.

This table lists Dynamic Parsing functions alphabetically by name, with descriptions. Click the function name to get information about it.

Function	Description
beginDynamicParse	Begins parsing a new message.
dynamicParse	Continues to parse data from a data buffer.
dynamicParseInputstream	Callback routine for parsing chunks of data from an input stream.
endDynamicParse	Continues to parse data from an input stream.
mimeDynamicParser_free	Frees the dynamic parser.
mimeDynamicParser_new	Creates a dynamic parser.

[Top] [MIME Functions by Functional Group]

beginDynamicParse

Begins parsing a new message.

Syntax

```
#include <mimeparser.h>
int beginDynamicParse( struct mimeParser * in_pParser );
```

Parameters The function has the following parameter:

in_pParser	Pointer to parser you are using.

Returns

- If successful, the function returns MIME_OK.

- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description For more information, see Dynamic Parsing.

See Also `mimeDynamicParser_new`

[Top] [Dynamic Parsing Functions] [MIME Functions by Functional Group]

---

dynamicParse

Continues to parse data from a data buffer.

Syntax

```
#include <mimeparser.h>
int dynamicParse( struct mimeParser *in_pParser,
             char * in_pData, int in_nLen );
```

Parameters The function has the following parameters:

`in_pParser`	Pointer to the parser.
`in_pData`	Pointer to the input stream.
`in_nLen`	Length of data to parse.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description For more information, see Dynamic Parsing.

See Also `dynamicParseInputstream`

[Top] [Dynamic Parsing Functions] [MIME Functions by Functional Group]

---

dynamicParseInputstream

Continues to parse data from an input-stream.

Syntax

```
#include <mimeparser.h>
int dynamicParseInputstream(
            struct mimeParser *in_pParser,
            struct nsmail_inputstream *in_pInput);
```

Parameters The function has the following parameters:

`in_pParser`	Pointer to the parser.
`in_pInput`	Pointer to data to parse.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description For more information, see Dynamic Parsing.

See Also `dynamicParse`

[Top] [Dynamic Parsing Functions] [MIME Functions by Functional Group]

---

endDynamicParse

Tells the parser that there is no more data to parse.

Syntax

```
#include <mimeparser.h>
int endDynamicParse( struct mimeParser *in_pParser );
```

Parameters The function has the following parameter:

`in_pParser`	Pointer to the parser.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description For more information, see Dynamic Parsing.

See Also `dynamicParseInputstream`

[Top] [Dynamic Parsing Functions] [MIME Functions by Functional Group]

---

mimeDynamicParser_free

Frees the dynamic parser.

Syntax

```
#include <mimeparser.h>
void mimeDynamicParser_free(
            struct mimeParser ** in_ppParser );
```

Parameters The function has the following parameter:

`in_ppParser`	Pointer to pointer to the parser.

Description For more information, see Dynamic Parsing.

See Also `beginDynamicParse, mime_basicPart_free_all, mime_multiPart_free_all, mime_message_free_all, mime_messagePart_free_all`

[Top] [Dynamic Parsing Functions] [MIME Functions by Functional Group]

---

mimeDynamicParser_new

Creates a dynamic parser.

Syntax

```
#include <mimeparser.h>
int mimeDynamicParser_new(
            mimeDataSink_t * in_pDataSink,
            struct mimeParser ** out_ppParser);
```

Parameters The function has the following parameters:

`in_pDataSink`	Pointer to data sink to use for callbacks; if no callbacks are made: `null`.
`out_ppParser`	Pointer to pointer to the dynamic parser you are using.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function creates the dynamic parser. If the parse operation is not going to use callbacks to a data sink, the value for `pdataSink` should be `null`.

For more information, see Dynamic Parsing and Creating the Dynamic Parser.

See Also `beginDynamicParse, mimeDynamicParser_free`

[Top] [Dynamic Parsing Functions] [MIME Functions by Functional Group]

Message Parsing Functions

These functions provide MIME parsing functions. For more information, see Parsing MIME Messages.

his table lists MIME Data Sink functions alphabetically by name, with descriptions. Click the function name to get information about it.

Function	Description
`parseEntireMessage`	Parses an entire message from a data buffer.
`parseEntireMessageInputstream`	Parse an entire message from an input stream.

[Top] [MIME Functions by Functional Group]

---

parseEntireMessage

Parses an entire message from a data buffer.

Syntax

```
#include <mimeparser.h> (
int parseEntireMessage (char * in_pData,
                  int in_nLen,
                  struct mime_message ** in_ppMimeMessage
```

Parameters The function has the following parameters:

`in_pData`	Pointer to the data buffer
`in_nLen`	Length of data to parse.
`in_ppMimeMessage`	Pointer to pointer to the MIME message.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function is one of two routines for parsing an entire message. `parseEntireMessage` parses data received from a data buffer, while `parseEntireMessageInputstream` parses an entire message from an input stream.

For more information, see Parsing the Entire Message.

See Also `parseEntireMessageInputstream`

[Top] [Message Parsing Functions] [MIME Functions by Functional Group]

---

parseEntireMessageInputstream

Parses an entire message in a single operation from an input stream.

Syntax

```
#include <mimeparser.h>
int parseEntireMessageInputstream(
         struct nsmail_inputstream *in_pInput,
         struct mime_message ** out_ppMimeMessage);
```

Parameters The function has the following parameters:

`in_pInput`	Pointer to the input stream
`out_ppMimeMessage`	Pointer to pointer to MIME message.

Returns

- If successful, the function returns `MIME_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function is one of two routines for parsing an entire message in a single operation. `parseEntireMessage` parses data received from a data buffer, while `parseEntireMessageInputstream` parses an entire message from an input stream.

For more information, see Parsing the Entire Message.

See Also `parseEntireMessage`

[Top] [Message Parsing Functions] [MIME Functions by Functional Group]

Data Sink Functions

These functions create and destroy the MIME data sink. For more information, see Creating a Data Sink.

This table lists MIME Data Sink functions alphabetically by name, with descriptions. Click the function name to get information about it.

Function	Description
`mimeDataSink_free`	Frees a MIME data sink and its data members.
`mimeDataSink_new`	Creates a new data sink.

[Top] [MIME Functions by Functional Group]

---

mimeDataSink_free

Frees a MIME data sink and its data members.

Syntax

```
#include <mimeparser.h>
void mimeDataSink_free( mimeDataSink_t ** pp );
```

Parameters

The function has the following parameter:

pp	Pointer to pointer to MIME data sink to free.

Returns

- If successful, the function returns MIME_OK.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description

This function frees all mimeDataSink function pointers and deallocates its storage space. The user must free any opaque data.

For more information, see Creating a Data Sink.

See Also

parseEntireMessage, mimeDataSink_t, mimeDataSink_new

[Top] [Data Sink Functions] [MIME Functions by Functional Group]

---

mimeDataSink_new

Creates a new data sink.

Syntax

```
#include <mimeparser.h>
int mimeDataSink_new( mimeDataSink_t **pp );
```

Parameters

The function has the following parameters:

pp	Pointer to pointer to MIME data sink.

Returns

- If successful, the function returns MIME_OK.
- If unsuccessful, the function returns an error message. For a complete list, see MIME Error Codes.

Description This function initializes all `mimeDataSink` function pointers and allocates space for the structure. The data sink is made up of function pointers and opaque data, which are set to `null` when the sink is initialized.

For more information, see Creating a Data Sink and SDK Response Sinks for C.

See Also `parseEntireMessage`, `mimeDataSink_t`, `mimeDataSink_free`

[Top] [Data Sink Functions] [MIME Functions by Functional Group]

MIME Structures

This section defines MIME data structures, listed in alphabetical order.

file_mime_type	mime_messagePart_t
mime_basicPart_t	mime_multiPart_t
mime_header_t	mime_outputstream_t
mime_inputstream_t	mimeDataSink_t
mime_message_t	

All MIME definitions are found in the `mime.h` and `mimeparser.h` files. The source file for each structure is noted in its reference entry.

[Top]

---

file_mime_type

Represents the file MIME type.

Syntax

```
#include <mime.h>
typedef struct file_mime_type
{
   mime_content_type  content_type;
   char *             content_subtype;
   char *             content_params;
   mime_encoding_type mime_encoding;
} file_mime_type;
```

Fields The structure has the following fields:

`content_type`	Content description.
`content_subtype`	Pointer to content subtypes. For information, see MIME Content Types.
`content_params`	Pointer to content parameters.
`mime_encoding`	Encoding type. For values, see MIME Encoding Types.

Description This structure contains `MIME` Content Type and MIME Encoding information, and is used only by the `getFileMIMEType` function.

See Also `getFileMIMEType`

[Top] [MIME Structures]

---

mime_basicPart_t

Common structure for leaf parts: text, audio, video, image and application.

Syntax

```
#include <mime.h>
typedef struct mime_basicPart
{
 /* Common to all parts */
 char * content_description;
 mime_disp_type content_disposition;
 char * content_disp_params;
 /* content disposition parameters */

 mime_content_type content_type;
 char * content_subtype;
 char * content_type_params;
 char * contentID;
 mime_header_t * extra_headers;
/* additional X- and Content-headers */

 /* Specific to basic part */
 char * contentMD5;
 mime_encoding_type encoding_type;

 /* Data that client cannot directly manipulate */
 void * pInternal;
} mime_basicPart_t;
```

Fields The structure has the following fields:

`content_description`	Pointer to content description.
`content_disposition`	Pointer to content disposition. For values, see MIME Disposition Types.
`content_disp_params`	Pointer to content disposition parameters.
`content_type`	Pointer to content types. For values, see MIME Content Types.
`content_subtype`	Pointer to content subtypes. For information, see MIME Content Types.
`content_type_params`	Pointer to content type parameters.
`contentID`	Pointer to content ID type.
`extra_headers`	Additional X- and Content-headers.
`contentMD5`	Pointer to content MD5.
`encoding_type`	Pointer to encoding type. For values, see MIME Encoding Types
`pInternal`	Pointer to data that the client cannot directly manipulate.

Description This structure represents the MIME basic part, which can be of text, audio, video, image, and application content types.

For more information, see Structure of a MIME Message and Adding Content to the Message. For functions that manipulate this structure, see Basic (Leaf) Body Part Functions.

See Also `mime_multiPart_t, mime_messagePart_t, mime_message_t`

[Top] [MIME Structures]

---

mime_header_t

Represents the MIME message header.

Syntax

```
#include <mime.h>
typedef struct mime_header
{
   char * name;
```

```
   char * value;
   struct mime_header * next;
} mime_header_t;
```

Fields The structure has the following fields:

`name`	Pointer to field name of the header name:value pair.
`value`	Pointer to field body of the header name:value pair.
`next`	Pointer to next header, if present.

Description This structure represents the RFC 822-compliant MIME message header. It consists of lines that describe the sender, subject, recipient, date, version of MIME in use, and a variety of other information, depending on the implementation in use.

For more information, see Structure of a MIME Message and Adding Message Headers.

See Also `mime_basicPart_t, mime_multiPart_t, mime_messagePart_t, mime_message_t`

[Top] [MIME Structures]

---

mime_inputstream_t

Represents the MIME input stream.

Syntax

```
#include <mime.h>
typedef struct nsmail_inputstream mime_inputstream_t;
```

See Also `nsmail_inputstream_t`, `mime_outputstream_t`

[Top] [MIME Structures]

---

mime_message_t

Represents the MIME message.

Syntax

```
#include <mime.h>
typedef struct mime_message
{
   mime_header_t * rfc822_headers;

   /* Data that client cannot directly manipulate */
   void * pInternal;
} mime_message_t;
```

Fields

The structure has the following fields:

`rfc822_headers`	Pointer to message headers that conform to RFC 822 requirements.
`pInternal`	Pointer to data that the client cannot directly manipulate.

Description

This structure represents an RFC 822-compliant MIME message.

For more information, see Structure of a MIME Message and Building the MIME Message. For functions that manipulate this structure, see Message Functions.

See Also

`mime_header_t, mime_basicPart_t, mime_multiPart_t, mime_messagePart_t`

[Top] [MIME Structures]

---

mime_messagePart_t

Represents the MIME message part.

Syntax

```
#include <mime.h>
typedef struct mime_messagePart
{
   /* Common to all parts */
   char * content_description;
   mime_disp_type content_disposition;
   char * content_disp_params;
```

```
/* content disposition parameters */
   mime_content_type content_type;
   char * content_subtype;
   char * content_type_params;
   char * contentID;
   mime_header_t * extra_headers;
   /* Additional X- and Content-headers */

   /* Data that client cannot directly manipulate */
   void * pInternal;
} mime_messagePart_t;
```

Fields The structure has the following fields:

`content_description`	Pointer to content description.
`content_disposition`	Content disposition. For values, see MIME Disposition Types.
`content_disp_params`	Pointer to content disposition parameters. For values, see MIME Disposition Types.
`content_type`	Content types. For values, see MIME Content Types.
`content_subtype`	Pointer to content subtypes.
`content_type_params`	Pointer to content type parameters.
`contentID`	Pointer to content ID.
`extra_headers`	Pointer to additional X- and Content-headers.
`pInternal`	Pointer to data that the client cannot directly manipulate.

Description This structure represents the MIME message part.

For more information, see Structure of a MIME Message and Building the MIME Message. For functions that manipulate this structure, see Message Part Functions.

See Also `mime_basicPart_t, mime_multiPart_t`

[Top] [MIME Structures]

mime_multiPart_t

Represents the MIME multipart content type.

Syntax

```
#include <mime.h>
typedef struct mime_multiPart
{
   /* Common to all parts */
   char * content_description;
   mime_disp_type content_disposition;
   char * content_disp_params;
      /* content disposition parameters */
   mime_content_type content_type;
   char * content_subtype;
   char * content_type_params;
   char * contentID;
   mime_header_t * extra_headers;
   /* Additional X- and Content-headers */

   /* Data that client cannot directly manipulate */
   void * pInternal;
} mime_multiPart_t;
```

Fields

The structure has the following fields:

`content_description`	Pointer to content description.
`content_disposition`	Pointer to content disposition. For values, see MIME Disposition Types.
`content_disp_params`	Pointer to content disposition parameters.
`content_type`	Pointer to content types. For values, see MIME Content Types.
`content_subtype`	Pointer to content subtypes.
`content_type_params`	Pointer to content type parameters.
`contentID`	Pointer to content ID.
`extra_headers`	Pointer to additional X- and Content-headers.
`pInternal`	Pointer to data that the client cannot directly manipulate.

Description

This structure represents the MIME Multipart content type, which is made up of one or more sub-body parts. The Multipart type has several subtypes that describe how the sub-parts relate to each other, including: mixed, alternative, digest, and parallel.

For more information, see Structure of a MIME Message and Building the MIME Message. For functions that manipulate this structure, see Multipart Functions.

See Also `mime_basicPart_t, mime_messagePart_t`

[Top] [MIME Structures]

---

mime_outputstream_t

Represents the MIME output stream.

Syntax

```
#include <mime.h>
typedef struct nsmail_outputstream mime_outputstream_t;
```

See Also `nsmail_outputstream_t, mime_inputstream_t`

[Top] [MIME Structures]

---

mimeDataSink_t

Represents the MIME data sink used for MIME parsing.

Syntax

```
#include <mimeparser.h>
typedef struct mimeDataSink
{
   void (*header)( mimeDataSinkPtr_t pSink,
      void *pCallbackObject,
      char *name, char *value ); /* mime headers */
   void (*contentType)( mimeDataSinkPtr_t pSink,
      void *pCallbackObject,
      int nContentType ); /* content type */
   void (*contentSubType)( mimeDataSinkPtr_t pSink,
      void *pCallbackObject,
      char * contentSubType ); /* content sub type */
   void (*contentTypeParams) ( mimeDataSinkPtr_t pSink,
      void *pCallbackObject,
```

```
      char * contentTypeParams );
      /* content type extra parameters */
   void (*contentID)( mimeDataSinkPtr_t pSink,
      void *pCallbackObject,
      char * contentID ); /* content ID */
   void (*contentMD5)( mimeDataSinkPtr_t pSink,
      void *pCallbackObject,
      char * contentMD5 ); /* content MD5 */
   void (*contentDisposition) ( mimeDataSinkPtr_t pSink,
      void *pCallbackObject,
      int nContentDisposition ); /* content disposition */
   void (*contentDispParams)(mimeDataSinkPtr_t pSink,
      void *pCallbackObject, char * contentDispParams ); /*
content disposition parameters */
   void (*contentDescription) ( mimeDataSinkPtr_t pSink,
      void *pCallbackObject,
      char * contentDescription ); /* content description */
   void (*contentEncoding) ( mimeDataSinkPtr_t pSink,
      void *pCallbackObject,
      int nContentEncoding );/* content encoding */
   void *(*startMessageLocalStorage) ( mimeDataSinkPtr_t pSink,
      struct mime_message *m );
      /* signal start of message, local storage version */
   void *(*startMessage)(mimeDataSinkPtr_t pSink);
      /* signal start of message */
   void (*endMessage) ( mimeDataSinkPtr_t pSink,
      void *pCallbackObject );
      /* signal end of message */
   void *(*startBasicPartLocalStorage)(
      mimeDataSinkPtr_t pSink,
      struct mime_basicPart *m );
      /* signal start of basic part, local storage version */
   void *(*startBasicPart)(mimeDataSinkPtr_t pSink);
      /* signal start of basic part */
   void (*bodyData) ( mimeDataSinkPtr_t pSink,
      void *pCallbackObject,
      char bodyData[], int len ); /* message data */
   void (*endBasicPart)( mimeDataSinkPtr_t pSink,
      void *pCallbackObject );
      /* signal end of basic part */
   void *(*startMultiPartLocalStorage)(
      mimeDataSinkPtr_t pSink, struct mime_multiPart *m );
      /* signal start of multipart, local storage version */
   void *(*startMultiPart)(mimeDataSinkPtr_t pSink);
      /* signal start of multipart */
   void (*boundary)( mimeDataSinkPtr_t pSink,
```

```
      void *pCallbackObject, char * boundary );
   void (*endMultiPart)( mimeDataSinkPtr_t pSink,
      void *pCallbackObject );
      /* signal end of multipart */
   void *(*startMessagePartLocalStorage)(
      mimeDataSinkPtr_t pSink,
      struct mime_messagePart *m );
      /* signal start of message part, local storage version */
   void *(*startMessagePart)(mimeDataSinkPtr_t pSink);
      /* signal start of message part */
   void (*endMessagePart)( mimeDataSinkPtr_t pSink,
      void *pCallbackObject );
      /* signal end of message part */
   } mimeDataSink_t;
```

Fields The structure has the following fields:

```
void (*header)(
   mimeDataSinkPtr_t pSink,
   void *pCallbackObject,
   char *name,
   char *value );
```

MIME headers. Parameters: Identifier of the data sink, identifier of the callback object, name and value portions of header's `name:value` pair.

```
void (*contentType)(
   mimeDataSinkPtr_t pSink,
   void *pCallbackObject,
   int nContentType );
```

Content type. Parameters: Identifier of the data sink, identifier of the callback object, content type of the body part.

```
void (*contentSubType)(
   mimeDataSinkPtr_t pSink,
   void *pCallbackObject,
   char * contentSubType );
```

Content sub type. Parameters: Identifier of the data sink, identifier of the callback object, content subtype of the body part.

```
void (*contentTypeParams)(
   mimeDataSinkPtr_t pSink,
   void *pCallbackObject,
   char * contentTypeParams );
```

Content type extra parameters. Parameters: Identifier of the data sink, identifier of the callback object, content type parameters.

```
void (*contentID)(
   mimeDataSinkPtr_t pSink,
   void *pCallbackObject,
   char * contentID );
```

Content ID. Parameters: Identifier of the data sink, identifier of the callback object, content identifier.

```
void (*contentMD5)(
   mimeDataSinkPtr_t pSink,
   void *pCallbackObject,
   char * contentMD5 );
```

Content MD5. Parameters: Identifier of the data sink, identifier of the callback object, contentMD5 of the body part.

```
void (*contentDisposition)(
   mimeDataSinkPtr_t pSink,
   void *pCallbackObject,
   int nContentDisposition );
```

Content disposition. Parameters: Identifier of the data sink, identifier of the callback object, content disposition.

`void (*contentDispParams)( mimeDataSinkPtr_t pSink, void *pCallbackObject, char * contentDispParams );`	Content disposition parameters. Parameters: Identifier of the data sink, identifier of the callback object, content disposition params.
`void (*contentDescription)( mimeDataSinkPtr_t pSink, void *pCallbackObject, char * contentDescription );`	Content description. Parameters: Identifier of the data sink, identifier of the callback object, content description of the body part.
`void (*contentEncoding)( mimeDataSinkPtr_t pSink, void *pCallbackObject, int nContentEncoding );`	Content encoding. Parameters: Identifier of the data sink, identifier of the callback object, encoding type for the message content.
`void *(*startMessageLocalStorage)( mimeDataSinkPtr_t pSink, struct mime_message *m );`	Signals start of message, local storage version. Parameters: Identifier of the data sink, MIME message that is starting.
`void *(*startMessage)( mimeDataSinkPtr_t pSink);`	Signals start of message. Parameter: Identifier of the data sink.
`void (*endMessage)( mimeDataSinkPtr_t pSink, void *pCallbackObject );`	Signals end of message. Parameters: Identifier of the data sink, identifier of the callback object
`void *( *startBasicPartLocalStorage)( mimeDataSinkPtr_t pSink, struct mime_basicPart *m );`	Signals start of basic part, local storage version. Parameters: Identifier of the data sink, MIME basic part to store.
`void *(*startBasicPart)( mimeDataSinkPtr_t pSink,);`	Signals start of basic part. Parameter: Identifier of the data sink.
`void (*bodyData)( mimeDataSinkPtr_t pSink, void *pCallbackObject, char bodyData[], int len );`	Message data. Parameters: Identifier of the data sink, identifier of the callback object, message part data, length of the data.
`void (*endBasicPart)( mimeDataSinkPtr_t pSink, void *pCallbackObject );`	Signals end of basic part. Parameters: Identifier of the data sink, identifier of the callback object.
`void *(*startMultiPartLocalStorage)( mimeDataSinkPtr_t pSink, struct mime_multiPart *m );`	Signals start of multipart, local storage version. Parameters: Identifier of the data sink, MIME multipart to store.
`void *(*startMultiPart)( mimeDataSinkPtr_t pSink);`	Signals start of multipart. Parameter: Identifier of the data sink.

`void (*boundary)( mimeDataSinkPtr_t pSink, void *pCallbackObject, char * boundary );`	Generates and returns a boundary string for use in multiparts and for other uses as required. Parameters: Identifier of the data sink, identifier of the callback object, the boundary string.
`void (*endMultiPart)( mimeDataSinkPtr_t pSink, void *pCallbackObject );`	Signals end of multipart. Parameters: Identifier of the data sink, identifier of the callback object.
`void *( *startMessagePartLocalStorage)( mimeDataSinkPtr_t pSink, struct mime_messagePart *m );`	Signals start of message part, local storage version. Parameters: Identifier of the data sink, MIME message part to store.
`void *(*startMessagePart)( mimeDataSinkPtr_t pSink);`	Signals start of message part. Parameter: Identifier of the data sink.
`void (*endMessagePart)( mimeDataSinkPtr_t pSink, void *pCallbackObject );`	Signals end of message part. Parameters: Identifier of the data sink, identifier of the callback object.

Description The MIME data sink structure is made up of function pointers and opaque data. You must define the functions yourself, and you must point these pointers to your functions. For more information, see SDK Response Sinks for C and Creating a Data Sink.

The function pointers serve as callbacks for many MIME functions. See MIME Data Sink Callbacks.

See Also `mimeDynamicParser_new`

[Top] [`mimeDataSink_t`] [MIME Structures]

MIME Definitions

These definitions are shared by MIME API functions.

- MIME Error Codes

- MIME Type Definitions
- MIME Buffer Size

All error code definitions are found in `mime.h`.

[Top]

MIME Error Codes

All MIME error code definitions are found in `mime.h`

Error Code	Value	Description
`MIME_OK`	0	Successful completion of function.
`MIME_ERR_UNINITIALIZED`	-1	Uninitialized parameter.
`MIME_ERR_INVALIDPARAM`	-2	Invalid parameters.
`MIME_ERR_OUTOFMEMORY`	-3	Out of memory.
`MIME_ERR_UNEXPECTED`	-4	Unexpected element.
`MIME_ERR_IO`	-5	Error in input/output operation.
`MIME_ERR_IO_READ`	-6	Error in reading input stream.
`MIME_ERR_IO_WRITE`	-7	Error in writing to output stream.
`MIME_ERR_IO_SOCKET`	-8	Socket connection error.
`MIME_ERR_IO_SELECT`	-9	Error in selecting message.
`MIME_ERR_IO_CONNECT`	-10	Error in connecting.
`MIME_ERR_IO_CLOSE`	-11	Error in closing.
`MIME_ERR_PARSE`	-12	Internal parsing error occurred.
`MIME_ERR_TIMEOUT`	-13	Timeout occurred. Recoverable error. Wait and call `processResponses` later.
`MIME_ERR_INVALID_INDEX`	-14	Invalid index.
`MIME_ERR_CANTOPENFILE`	-15	Cannot open file.
`MIME_ERR_CANT_SET`	-16	Cannot set MIME text.

Error Code	Value	Description
MIME_ERR_ALREADY_SET	-17	Item already set.
MIME_ERR_CANT_DELETE	-18	Cannot delete item.
MIME_ERR_CANT_ADD	-19	Cannot add item.
MIME_ERR_EOF	-1	End of file reached.
MIME_ERR_EMPTY_DATASINK	-83	Data sink missing.
MIME_ERR_ENCODE	-84	Encoding error.
MIME_ERR_NO_SUCH_HEADER	-85	Header type does not exist.
MIME_ERR_NO_HEADERS	-86	Header missing.
MIME_ERR_NOT_SET	-87	Item not set.
MIME_ERR_NO_BODY	-88	Message body missing.
MIME_ERR_NOT_FOUND	-89	Cannot find search item.
MIME_ERR_NO_CONTENT_SUBTYPE	-90	Content subtype missing.
MIME_ERR_INVALID_ENCODING	-91	Invalid encoding type.
MIME_ERR_INVALID_BASICPART	-92	Invalid basic message part.
MIME_ERR_INVALID_MULTIPART	-93	Invalid multipart.
MIME_ERR_INVALID_MESSAGEPART	-94	Invalid message part.
MIME_ERR_INVALID_MESSAGE	-95	Invalid message.
MIME_ERR_INVALID_CONTENTTYPE	-96	Invalid content type.
MIME_ERR_INVALID_CONTENTID	-97	Invalid content identifier.
MIME_ERR_NO_DATA	-98	Data missing.
MIME_ERR_NOTIMPL	-99	Function or feature not implemented.

[Top] [MIME Definitions]

MIME Type Definitions

- MIME Disposition Types
- MIME Encoding Types
- MIME Content Types

All C interface definitions are found in `mime.h` and `mimeparser.h`.

[Top] [MIME Definitions]

MIME Encoding Types

MIME Encoding Type definitions are found in `mime.h`._

MIME Encoding Type	Description
`MIME_ENCODING_BASE64`	Base 64 encoding.
`MIME_ENCODING_QP`	Quoted Printable encoding.
`MIME_ENCODING_Q`	Quoted encoding.
`MIME_ENCODING_7BIT`	7 bit data with NO transfer encoding.
`MIME_ENCODING_8BIT`	8 bit data with NO transfer encoding.
`MIME_ENCODING_BINARY`	Binary data with NO transfer encoding.

[Top] [MIME Definitions] [MIME Type Definitions]

MIME Disposition Types

MIME Disposition Type definitions are found in `mime.h`.

MIME Display Type	Description
`MIME_DISPOSITION_UNINITIALIZED = 0`	Setting not in use.

MIME Display Type	Description
MIME_DISPOSITION_INLINE = 1	Display attachment in-line within message.
MIME_DISPOSITION_ATTACHMENT	Access attachment from icon.

[Top] [MIME Definitions] [MIME Type Definitions]

MIME Content Types

MIME Content Type definitions are found in mime.h.

MIME Content Type	Values	Description
MIME_CONTENT_TEXT	1	Text content.
MIME_CONTENT_AUDIO	2	Audio content.
MIME_CONTENT_IMAGE	3	Image content.
MIME_CONTENT_VIDEO	4	Video content.
MIME_CONTENT_APPLICATION	4	Application content.
MIME_CONTENT_MULTIPART	5	Multipart content.
MIME_CONTENT_MESSAGEPART	6	Message content.

[Top] [MIME Definitions] [MIME Type Definitions]

MIME Buffer Size

MIME Buffer Size definitions are found in mime.h.

Buffer Definition	Value	Description
MIME_BUFSIZE	1024	Default buffer size for MIME.

[Top] [MIME Definitions]

Chapter

11

IMAP C API Reference

This chapter describes the functions and structures of the C language API for the IMAP4 (Internet Message Access Protocol, Version 4) protocol of the Messaging Access SDK.

- IMAP4 Functions by Functional Group
- IMAP4 Functions by Name
- IMAP4 Structures

You'll find links to each function and structure in this introduction. Each reference entry gives the name of the function or structure, its header file, its syntax, and its parameters.

All C interface definitions are found in the `imap4.h` file.

[Top]

IMAP4 Functions by Functional Group

IMAP4 functions are organized into functional groups. Click the group name to find the reference entries for the functions in the group. To find functions listed in alphabetical order by name, see IMAP4 Functions by Name.

- General Functions
- Non-Authenticated State Functions
- Authenticated State Functions
- Selected State Functions
- Extended IMAP4 Functions

[Top] [IMAP4 Functions by Task]

IMAP4 Functions by Name

This table lists IMAP4 functions in alphabetical order by name. To find functions listed by functional group, see IMAP4 Functions by Functional Group.

imap4_append	imap4_namespace
imap4_capability	imap4_noop
imap4_check	imap4_processResponses
imap4_close	imap4_rename
imap4_copy	imap4_select
imap4_connect	imap4_sendCommand
imap4_create	imap4_setChunkSize
imap4_delete	imap4_set_option
imap4_deleteACL	imap4_setResponseSink
imap4_disconnect	imap4_setTimeout
imap4_examine	imap4_search

imap4_expunge	imap4_setACL
imap4_fetch	imap4Sink_initialize
imap4_free	imap4Sink_free
imap4_getACL	imap4_status
imap4_get_option	imap4_store
imap4_initialize	imap4_subscribe
imap4_list	imap4Tag_free
imap4_listRights	imap4_unsubscribe
imap4_lsub	imap4_uidCopy
imap4_login	imap4_uidFetch
imap4_logout	imap4_uidSearch
imap4_myRights	imap4_uidStore

[Top] [IMAP4 Functions by Name] [IMAP4 Functions by Task]

General Functions

This table lists IMAP4 General functions, which map to IMAP4 protocol commands defined in RFC 2060. For links to IMAP4 RFCs, see IMAP4 RFCs. Click the function name to get information about it.

Function	Description
imap4_connect	Connects to the IMAP server in_host through the specified port.
imap4_disconnect	Disconnects from the IMAP4 server.
imap4_free	Frees the IMAP4 client structure and its data members.
imap4_get_option	Gets the IO model.
imap4_initialize	Allocates space for the IMAP4 client structure.
imap4_processResponses	Processes the server responses for all API commands executed prior to this command.
imap4_sendCommand	Sends the specified command to the IMAP4 server.

Function	Description
imap4_setChunkSize	Sets the size of the message data chunk returned in fetchData.
imap4_set_option	Sets the IO model.
imap4_setResponseSink	Sets the response sink to the specified sink.
imap4_setTimeout	Sets the amount of time allowed to wait for data within processsResponses before returning control to the user.
imap4Sink_free	Frees the response sink.
imap4Sink_initialize	Initializes all pointers to default pointers (empty function definitions).
imap4Tag_free	Frees the tag associated with a command.

[Top] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

imap4_connect

Connects to the IMAP server in_host through the port in_portNumber.

Syntax

```
#include <imap4.h>
int imap4_connect( imap4Client_t * in_pimap4,
                   const char * in_host,
                   unsigned short in_port,
                   char** out_ppTagID);
```

Parameters The function has the following parameters:

in_pimap4	Pointer to the IMAP4 client.
in_host	Pointer to the name of the host to which to connect.
in_port	Number of port to which you want to connect.
out_ppTagID	Pointer to pointer to the tag associated with the function. Use imap4Tag_free to free the associated memory.

Returns

- If successful, the function returns NSMAIL_OK.

- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description For more information, see Connecting to a Server.

This function is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

See Also `imap4Sink_t.error`, `imap4Sink_t.ok`, `imap4_login`, `imap4Tag_free`, `imap4_disconnect`

[Top] [General Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_disconnect

Disconnects from the IMAP4 server.

Syntax

```
#include <imap4.h>
int imap4_disconnect( imap4Client_t * in_pimap4 );
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client to disconnect.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function closes the socket connection with the server. For more information, see Connecting to a Server.

This function is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

See Also `imap4Sink_t.bye`, `imap4_connect`, `imap4_logout`, `imap4_close`

[Top] [General Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

imap4_free

Frees the IMAP4 client structure and its underlying data members.

Syntax

```
#include <imap4.h>
int imap4_free(imap4Client_t ** in_ppimap4);
```

Parameters The function has the following parameters:

`in_ppimap4` Pointer to a pointer to a client structure associated with the data members to free. Sets the pointer to the client structure to `null` on return.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function frees the `imap4Client_t` structure. When the function returns, the client structure is set to `null`. The user must free any pointers to opaque data.

See Also `imap4_logout, imap4_initialize, imap4Sink_free`

[Top] [General Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

imap4_get_option

Gets the IO model.

Syntax

```
#include <imap4.h>
int imap4_get_option(imap4Client_t * in_pimap4,
                     int in_option,
                     void* in_pOptionData);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_option`	Pointer to option you want to set. See `nsmail_io_fns_t`. For the option that is currently supported, see "Option Definition."
`in_pOptionData`	Pointer to option data set with `imap4_set_option`.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

See Also `imap4_set_option`

[Top] [General Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_initialize

Initializes and allocates the `IMAP4` client structure and sets the response sink.

Syntax

```
#include <imap4.h>
int imap4_initialize(imap4Client_t ** in_ppimap4,
                imap4Sink_t * in_pimap4Sink);
```

Parameters The function has the following parameters:

`in_ppimap4`	Pointer to pointer to the IMAP4 client.
`in_pimap4Sink`	Pointer to an existing IMAP4 response sink.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description For more information, see Creating a Client.

See Also `imap4Sink_free`

[Top] [General Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

imap4_processResponses

Processes the server responses for all API commands executed prior to this command.

Syntax

```
#include <imap4.h>
int imap4_processResponses(imap4Client_t* in_pimap4);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If a time-out occurs, the function returns `NSMAIL_ERR_TIMEOUT`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function processes the server responses for API commands. It invokes the callback functions provided by the user for all responses that are available at the time of execution.

If a time-out occurs, the user can continue by calling `imap4_processResponses` again.

Do not call `imap4_processResponses` after these operations: disconnecting, the set functions, initializing and freeing the client and sink.

See Also `imap4Sink_t`

[Top] [General Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

imap4_sendCommand

Sends the specified command to the IMAP4 server.

Syntax

```
#include <imap4.h>
int imap4_sendCommand(imap4Client_t* in_pimap4,
                const char* in_command,
                char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_command`	Pointer to the unsupported command to send.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description You can use this function to extend the protocol to meet your application needs. Using this function, you can extend the functionality of the SDK to meet your needs by adding functions or passing different parameters to a function.

This function is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

See Also `imap4Sink_t.rawResponse`, `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4Tag_free`

[Top] [General Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_setChunkSize

Sets the size of the message data chunk returned in `fetchData`.

Syntax

```
#include <imap4.h>
int imap4_setChunkSize(imap4Client_t* in_pIMAP4,
```

```
                                int in_size);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_size`	Size of chunk to set.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

See Also `imap4_setTimeout`, `imap4_connect`

[Top] [General Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_set_option

Sets the IO model.

Syntax

```
#include <imap4.h>
int imap4_set_option(imap4Client_t* in_pimap4,
                     int in_option,
                     void* in_pOptionData);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_option`	Pointer to option you want to set. See `nsmail_io_fns_t`. For the option that is currently supported, see "Option Definition."
`in_pOptionData`	Pointer to option data.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

See Also `imap4_get_option`

[Top] [General Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

imap4_setResponseSink

Sets the sink to the specified sink.

Syntax

```
#include <imap4.h>
void imap4_setResponseSink(imap4Client_t* in_pimap4,
                           imap4Sink_t *in_pimap4Sink);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_pimap4Sink`	Pointer to the response sink to set.

Description This function sets the sink to `in_pimap4Sink`. This response sink replaces the one that was passed into `imap4_initialize`.

See Also `imap4_initialize`

[Top] [General Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

imap4_setTimeout

Sets the amount of time allowed to wait for data within `processsResponses` before returning control to the user.

Syntax

```
include <imap4.h>
void imap4_setTimeout(imap4Client_t* in_pimap4,
                      int in_timeout);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_timeout`	Time-out period to set. Values, in seconds: • -1 = infinite time-out (default) • 0 = no waiting • >0 = length of time-out period

Description This function sets the amount of time, in seconds, allowed to wait before returning control to the user.

See Also `imap4_processResponses`

[Top] [General Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4Sink_free

Frees the IMAP4 response sink and its data members.

Syntax

```
#include <imap4.h>
int imap4Sink_free(imap4Sink_t** out_ppimap4Sink);
```

Parameters The function has the following parameters:

`out_ppimap4Sink`	Pointer to a pointer to a client structure associated with the response sink to free.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function frees the IMAP4 response sink. It sets the pointer to the sink structure to `null` on return. The user must free any opaque data.

See Also `imap4Sink_initialize, imap4Tag_free, imap4_free`

[Top] [General Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

imap4Sink_initialize

Allocates space for the IMAP4 sink structure.

Syntax

```
#include <imap4.h>
int imap4Sink_initialize(imap4Sink_t** out_ppimap4Sink);
```

Parameters

The function has the following parameters:

`out_ppimap4Sink` Pointer to the IMAP4 client.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description

This function initializes all pointers to default pointers (empty function definitions) and allocates the space for the `imap4Sink` structure.

For more information, see Creating a Response Sink and SDK Response Sinks for C.

See Also

`imap4Sink_t, imap4_connect, imap4_noop, imap4Sink_free, imap4_initialize`

[Top] [General Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

imap4Tag_free

Frees the tag associated with an IMAP4 command.

Syntax

```
#include <imap4.h>
int imap4Tag_free(char** out_ppimap4Tag);
```

Parameters The function has the following parameters:

`out_ppimap4Tag` Pointer to pointer to the tag to free.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description Many functions allocate tags that can be associated with an `IMAP4` command. Any function that allocates a tag must use `imap4Tag_free` to free the memory associated with the tag. This function frees the tag and sets the tag pointer to `null` on return.

The tag (`out_ppTagID`) generated by the IMAP4 functions can be used to help match the command and the response associated with it within the `imap4Sink_t.taggedLine` response.

See Also `imap4_logout, imap4Sink_free, imap4_free, imap4Sink_t.taggedLine`

[Top] [General Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

Non-Authenticated State Functions

Non-Authenticated State functions can occur before login. For links to IMAP4 RFCs, see IMAP4 RFCs. Click the function name to get information about it.

Function	Description
`imap4_capability`	Requests a listing of capabilities that the server supports.
`imap4_login`	Identifies the client to the server with the user password.
`imap4_logout`	Informs the server that the client is done with the connection.
`imap4_noop`	Issues a command that always succeeds and does nothing.

[Top] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

imap4_capability

Requests a listing of capabilities that the server supports.

Syntax

```
#include <imap4.h>
int imap4_capability(imap4Client_t* in_pimap4,
                     char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function returns a listing of the IMAP4 extensions and authentication mechanisms that are supported by the server. It sends the CAPABILITY IMAP4 protocol command. For more information, see Determining Server Capabilities.

This function is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

See Also `imap4Sink_t.capability`, `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4_connect`, `imap4Tag_free`

[Top] [Non-Authenticated State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

imap4_login

Identifies the client to the server and carries the plain text password that authenticates this user.

Syntax

```
#include <imap4.h>
int imap4_login(imap4Client_t* in_pimap4,
             const char* in_user,
             const char* in_password,
             char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_user`	Pointer to the user name.
`in_password`	Pointer to the user password.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function identifies the client to the server and carries the plain text password that authenticates this user. It sends the LOGIN IMAP4 protocol command. For more information, see Logging In and Out.

This function is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

See Also `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4_logout`, `imap4Tag_free`

[Top] [Non-Authenticated State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

imap4_logout

Informs the server that the client is done with the connection.

Syntax

```
#include <imap4.h>
int imap4_logout(imap4Client_t* in_pimap4,
             char** out_ppTagID );
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function closes the connection and logs the user out. It sends the LOGOUT IMAP4 protocol command. For more information, see Logging In and Out.

This function is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

See Also `imap4Sink_t.bye`, `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4_close`, `imap4Tag_free`

[Top] [Non-Authenticated State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_noop

Issues a command that always succeeds and does nothing.

Syntax

```
#include <imap4.h>
int imap4_noop(imap4Client_t* in_pimap4,
            char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function issues a command that should always succeed. it sends the NOOP IMAP4 protocol command. In itself, `imap4_noop` does nothing, but it induces useful side effects.

Most IMAP4 servers check for messages whenever a command is issued. In the absence of commands, the server does not check for messages and may disconnect. To keep the server open indefinitely and check for messages periodically, the developer could call `imap4_noop` at set intervals.

The `imap4_noop` function does nothing in itself, so it is ideal for polling for new mail and ensuring that the server connection is still active. `imap4_noop` resets the autologout timer inside the server and results in the retrieval of unsolicited server responses. This may indicate the arrival of new messages or a change in the attributes of an existing message. For an example, see Checking for New Messages.

This function is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

Note Remember to use `imap4Tag_free` to free the memory associated with the tag allocated by this function. §

See Also `imap4Sink_t.exists`, `imap4Sink_t.expunge`, `imap4Sink_t.recent`, `imap4Sink_t.fetchStart`, `imap4Sink_t.fetchFlags`, `imap4Sink_t.fetchEnd`, `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4_connect`, `imap4Tag_free`

[Top] [Non-Authenticated State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

Authenticated State Functions

Authenticated State functions provide operations involving mailboxes. Before these functions are available, login must take place.

Function	Description
imap4_append	Appends a message to the specified mailbox.
imap4_create	Creates a mailbox with the given name.
imap4_delete	PERMANENTLY removes the mailbox with the given name.
imap4_examine	Selects a server mailbox for read-only access.
imap4_list	Lists the mailboxes.
imap4_lsub	Returns a subset of user-defined “active” or “subscribed” names.
imap4_rename	Renames a mailbox.
imap4_select	Selects a mailbox on the server.
imap4_status	Request the status of a particular mailbox.
imap4_subscribe	Adds the specified mailbox name to the server's set of “active” or “subscribed” mailboxes.
imap4_unsubscribe	Removes a mailbox from server's subscribed mailbox list.

[Top] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

imap4_append

Appends a message to the specified mailbox.

Syntax

```
#include <imap4.h>
int imap4_append( imap4Client_t* in_pimap4,
                  const char* in_mailbox,
                  const char* in_optFlags,
                  const char* in_optDateTime,
```

```
                    nsmail_inputstream_t* in_literal,
                    char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_mailbox`	Pointer to the name of mailbox to which to append message.
`in_optFlags`	Pointer to a list of optional flags in parentheses.
`in_optDateTime`	Pointer to optional date/time string.
`in_pInputLiteralStream`	Pointer to the message literal.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function appends a literal argument as a new message to the end of the specified mailbox.

This function sends the APPEND IMAP4 protocol command. It is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

See Also `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4_noop`, `imap4Tag_free`

[Top] [Authenticated State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_create

Creates a mailbox with the given name.

Syntax

```
#include <imap4.h>
int imap4_create( imap4Client_t* in_pimap4,
                  const char* in_mailbox,
```

```
                    char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_mailbox`	Pointer to the name of mailbox to create.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function creates a mailbox with the given name. If you want to create a directory on the server side, be careful to use the correct separator character.

This function sends the CREATE IMAP4 protocol command. It is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

See Also `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4_delete`, `imap4_rename`, `imap4_subscribe`, `imap4_unsubscribe`, `imap4_list`, `imap4_lsub`, `imap4Tag_free`

[Top] [Authenticated State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_delete

PERMANENTLY removes the mailbox with the given name.

Syntax

```
#include <imap4.h>
int imap4_delete( imap4Client_t* in_pimap4,
                  const char* in_mailbox,
                  char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_mailbox`	Pointer to the name of mailbox to delete.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function PERMANENTLY removes the mailbox with the given name. You cannot and should not DELETE the INBOX. The system needs it.

This function sends the DELETE IMAP4 protocol command. It is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

See Also `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4_store`, `imap4_create`, `imap4_rename`, `imap4_examine`, `imap4_subscribe`, `imap4_unsubscribe`, `imap4_list`, `imap4_lsub`, `imap4Tag_free`

[Top] [Authenticated State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_examine

Selects a server mailbox for read-only access.

Syntax

```
#include <imap4.h>
int imap4_examine(imap4Client_t* in_pimap4,
                  const char* in_mailbox,
                  char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_mailbox`	Pointer to the name of mailbox to examine.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description When issued in the Authorization state, this function sends the EXAMINE IMAP4 protocol command, which moves the IMAP4 session into the Selected state. This function selects a mailbox on the server and makes it available for operations that involve read-only access to messages, such as checking, closing, searching, fetching, and copying.

The `imap4_select` function works like `imap4_examine`, except that it selects the mailbox and its messages with update access. The user can perform operations that permanently alter the mailbox, such as storing or expunging.

This function is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

See Also `imap4Sink_t.flags`, `imap4Sink_t.exists`, `imap4Sink_t.recent`, `imap4Sink_t.ok`, `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4_select`, `imap4Tag_free`

[Top] [Authenticated State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_list

Lists the names of mailboxes matching the specified pattern.

Syntax

```
#include <imap4.h>
int imap4_list(imap4Client_t* in_pimap4,
               const char* in_refName,
               const char* in_mailbox,
```

```
                    char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_refName`	Pointer to the mailbox name within which you want to search.
`in_mailbox`	Pointer to a mailbox name with possible wild cards.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function lists a subset of all names available to the client and returns the tag associated with this function. The names listed must match the pattern given in the `in_mailbox` and `in_refName` parameters.

This function sends the LIST IMAP4 protocol command. It is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

See Also `imap4Sink_t.list`, `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4_lsub`, `imap4_create`, `imap4_delete`, `imap4_rename`, `imap4_subscribe`, `imap4_unsubscribe`, `imap4_lsub`, `imap4Tag_free`

[Top] [Authenticated State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_lsub

Returns a subset of user-defined "active" or "subscribed." names.

Syntax

```
#include <imap4.h>
int imap4_lsub( imap4Client_t* in_pimap4,
            const char* in_refName,
            const char* in_mailbox,
            char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_refName`	Pointer to the reference name.
`in_mailbox`	Pointer to a mailbox name with possible wild cards.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function returns a subset of names from the set of names that the user has declared as "active" or "subscribed."

This function sends the LSUB IMAP4 protocol command. It is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

Note Use `imap4Tag_free` to free the associated tag. §

See Also `imap4Sink_t.lsub`, `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4_list`, `imap4_create`, `imap4_delete`, `imap4_rename`, `imap4_subscribe`, `imap4_unsubscribe`, `imap4_lsub`, `imap4Tag_free`

[Top] [Authenticated State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_rename

Renames a mailbox.

Syntax

```
#include <imap4.h>
int imap4_rename( imap4Client_t* in_pimap4,
                  const char* in_currentMB,
                  const char* in_newMB,
                  char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_currentMB`	Pointer to the mailbox's current name.
`in_newMB`	Pointer to the mailbox's new name.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function sends the RENAME IMAP4 protocol command. It is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

Note Use `imap4Tag_free` to free the associated tag. §

See Also `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4_select`, `imap4_create`, `imap4_delete`, `imap4_subscribe`, `imap4_unsubscribe`, `imap4_list`, `imap4_lsub`, `imap4Tag_free`

[Top] [Authenticated State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_select

Selects a mailbox on the server.

Syntax

```
#include <imap4.h>
int imap4_select( imap4Client_t* in_pimap4,
             const char* in_mailbox,
             char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_mailbox`	Pointer to the name of mailbox to select.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description When issued in the Authorization state, this function sends the SELECT IMAP4 protocol command, which moves the IMAP4 session into the Selected state. This function selects a mailbox on the server and makes it available for operations that involve messages, such as checking, closing, searching, fetching, copying, storing, and expunging.

The `imap4_examine` function works like `imap4_select`, except that it selects the mailbox and its messages with read-only access. Operations that permanently alter the mailbox, such as storing, are not available.

This function is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

Note Use `imap4Tag_free` to free the associated tag. §

See Also `imap4Sink_t.flags`, `imap4Sink_t.exists`, `imap4Sink_t.recent`, `imap4Sink_t.ok`, `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4_examine`, `imap4Tag_free`

[Top] [Authenticated State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_status

Requests the status of the indicated mailbox.

Syntax

```
#include <imap4.h>
```

```
int imap4_status(imap4Client_t* in_pimap4,
            const char* in_mailbox,
            const char* in_statusData,
            char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_mailbox_name`	Pointer to the name of mailbox for which to get status.
`in_statusData`	Pointer to status data: All or a subset of `MESSAGES, RECENT, UID_NEXT, UID_VALIDITY, UNSEEN`.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function requests one or more types of status from the specified mailbox. It sends the STATUS IMAP4 protocol command.

This function is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

Note Use `imap4Tag_free` to free the associated tag. §

See Also `imap4Sink_t.statusMessages`, `imap4Sink_t.statusRecent`, `imap4Sink_t.statusUidnext`, `imap4Sink_t.statusUidvalidity`, `imap4Sink_t.statusUnseen`, `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4Tag_free`

[Top] [Authenticated State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_subscribe

Adds the specified mailbox name to the server's set of "active" or "subscribed" mailboxes.

Syntax

```
#include <imap4.h>
int imap4_subscribe( imap4Client_t* in_pimap4,
                const char* in_mailbox,
                char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_mailbox`	Pointer to the name of mailbox to which to add subscriber.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function sends the SUBSCRIBE IMAP4 protocol command.

To access the list of the server's set of "active" or "subscribed" mailboxes, use the `LSUB` protocol command through the `imap4_lsub` function.

This function is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

Note Use `imap4Tag_free` to free the associated tag. §

See Also `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4_lsub`, `imap4_create`, `imap4_delete`, `imap4_rename`, `imap4_unsubscribe`, `imap4_list`, `imap4Tag_free`

[Top] [Authenticated State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

imap4_unsubscribe

Removes mailbox from server's subscribed mailbox list.

Syntax

```
#include <imap4.h>
int imap4_unsubscribe(imap4Client_t* in_pimap4,
```

```
                    const char* in_mailbox,
                    char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_mailbox`	Pointer to the name of mailbox to remove from subscribed list.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function removes the specified mailbox name from the server's set of "active" or "subscribed" mailboxes. To do this, it sends the UNSUBSCRIBE IMAP4 protocol command.

Note Use `imap4Tag_free` to free the associated tag. §

This function is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

See Also `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4_subscribe`, `imap4_create`, `imap4_delete`, `imap4_rename`, `imap4_list`, `imap4_lsub`, `imap4Tag_free`

[Top] [Authenticated State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

Selected State Functions

Selected State functions provide operations involving messages. Before these functions are available, login must take place and the mailbox must be selected. For links to IMAP4 RFCs, see IMAP4 RFCs. Click the function name to get information about it.

Function	Description
imap4_check	Requests a checkpoint of the currently selected mailbox.
imap4_close	Closes the mailbox.
imap4_copy	Copies the specified message(s) from the currently selected mailbox to the end of the specified destination mailbox.
imap4_expunge	Removes all messages flagged for deletion.
imap4_fetch	Retrieves data specified by the fetch criteria for the given message set and returns the corresponding message sequence numbers.
imap4_search	Search the mailbox for messages that match the given search criteria and retrieve the corresponding message sequence numbers.
imap4_store	Alters data associated with a message in the mailbox.
imap4_uidCopy	Copies messages specified with a unique identifier.
imap4_uidFetch	Fetches messages specified with a unique identifier and retrieves the corresponding unique identifiers.
imap4_uidSearch	Searches the mailbox for messages that match your search criteria and retrieve the corresponding unique identifiers.
imap4_uidStore	Updates message data associated with a unique identifier.

[Top] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

imap4_check

Requests a checkpoint of the currently selected mailbox.

Syntax

```
#include <imap4.h>
int imap4_check(imap4Client_t* in_pimap4,
             char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function requests a checkpoint of the currently selected mailbox and flushes existing mailbox states to disk. Function behavior may be different, depending on the server implementation. It returns a tag.

This function sends the CHECK IMAP4 protocol command. It is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

Note Use `imap4Tag_free` to free the associated tag. §

See Also `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4Tag_free`

[Top] [Selected State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_close

Closes the mailbox.

Syntax

```
#include <imap4.h>
```

```
int imap4_close(imap4Client_t* in_pimap4,
                char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function closes a mailbox and removes any messages marked with the `\Deleted` flag. The session moves to the mailbox that contains the closed mailbox. This function sends the CLOSE IMAP4 protocol command. If you need to remove messages without closing, use `imap4_expunge`. Also see Closing a Mailbox.

This function is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

Note Use `imap4Tag_free` to free the associated tag. §

See Also `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4_logout`, `imap4_expunge`, `imap4Tag_free`

[Top] [Selected State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_copy

Copies the specified message(s) from the currently selected mailbox to the end of the specified destination mailbox.

Syntax

```
#include <imap4.h>
int imap4_copy( imap4Client_t* in_pimap4,
                const char* in_msgSet,
                const char* in_mailbox,
                char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_msgSet`	Pointer to the message numbers to copy, for example, “1:2”.
`in_mailbox`	Pointer to the name of target mailbox.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see “Error Definitions.”

Description This function sends the COPY IMAP4 protocol command. It is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

Note Use `imap4Tag_free` to free the associated tag. §

See Also `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4_store`, `imap4_expunge`, `imap4Tag_free`

[Top] [Selected State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_expunge

Removes all messages flagged for deletion.

Syntax

```
#include <imap4.h>
int imap4_expunge(imap4Client_t* in_pimap4,
                  char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns NSMAIL_OK.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function removes all messages in the mailbox that have the \Deleted flag. As described in RFC 2060, when a message is expunged, all messages with higher numbers are automatically decremented by one. This happens BEFORE this call returns, so, immediately following this call, you should decide whether you need to make adjustments to this change in the state of the message cache. Also see Closing a Mailbox.

This function sends the EXPUNGE IMAP4 protocol command. It is mapped to one or more callbacks in the imap4Sink_t structure. For more information, see IMAP4 Function Callback Mapping.

Note Use imap4Tag_free to free the associated tag. §

See Also imap4Sink_t.taggedLine, imap4Sink_t.error, imap4Sink_t.expunge, imap4_delete, imap4_close, imap4_store, imap4_copy, imap4Tag_free

[Top] [Selected State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_fetch

Retrieves data specified by the fetch criteria for the given message set and returns the corresponding message sequence numbers.

Syntax

```
#include <imap4.h>
int imap4_fetch(imap4Client_t* in_pimap4,
             const char* in_msgSet,
             const char* in_fetchCriteria,
             char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_msgSet`	Pointer to the message set.
`in_fetchCriteria`	Pointer to the criteria that fetched messages must meet. See the RFC 2060 Fetch Data Items table below for RFC 2060 values for this parameter.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function retrieves the data specified by the fetch criteria for the given message set. You can specify a variety of fetch criteria, as shown in Table 11.1 in this section. These fetch criteria are defined in RFC 2060. If you specify more than one item, place the items in a parenthesized list. This function sends the FETCH IMAP4 protocol command.

For more information, see Fetching Message Data.

Table 11.1 RFC 2060 Fetch Data Items

Data Item	Description
`ALL`	Combines `FLAGS, INTERNALDATE, ENVELOPE, RFC822.SIZE`
`BODY`	`[MIME-IMB]` body structure of the message, not non-extensible
`BODY[<section>]<<partial>>`	text of a particular body section; sets `\Seen` flag. `section`: zero or more part specifiers separated by periods, for example `HEADER, HEADER.FIELDS`. `partial`: whether fetch is partial
`BODY.PEEK[<section>]<<partial>>`	alternate to `BODY[<section>]`; does not set `\Seen` flag
`BODYSTRUCTURE`	`[MIME-IMB]` body structure of the message,
`ENVELOPE`	message's envelope structure
`FAST`	combines `FLAGS, INTERNALDATE, RFC822.SIZE`
`FLAGS`	flags set for this message

Data Item	Description
FULL	combines FLAGS, INTERNALDATE, RFC822.SIZE, ENVELOPE, BODY
INTERNALDATE	message's internal data
RFC822	equivalent to BODY[], but with different syntax in the resulting untagged FETCH data
RFC822.HEADER	equivalent to BODY.PEEK[HEADER], but with different syntax in the resulting untagged FETCH data
RFC822.SIZE	message size
RFC822.TEXT	equivalent to BODY[TEXT], but with different syntax in the resulting untagged FETCH
UID	unique identifier of the message

This function is mapped to one or more callbacks in the imap4Sink_t structure. For more information, see IMAP4 Function Callback Mapping.

Note Use imap4Tag_free to free the associated tag. §

See Also imap4Sink_t.fetchStart, imap4Sink_t.fetchEnd, imap4Sink_t.fetchSize, imap4Sink_t.fetchData, imap4Sink_t.fetchFlags, imap4Sink_t.fetchBodyStructure, imap4Sink_t.fetchEnvelope, imap4Sink_t.fetchInternalDate, imap4Sink_t.fetchHeader, imap4Sink_t.taggedLine, imap4Sink_t.error, imap4_search, imap4_uidFetch, imap4Tag_free

[Top] [Selected State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

imap4_search

Search the selected mailbox for messages that match your search criteria and retrieve the corresponding message sequence numbers.

Syntax

```
#include <imap4.h>
```

```
int imap4_search( imap4Client_t* in_pimap4,
             const char* in_criteria,
             char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_criteria`	Pointer to search criteria consisting of one or more search keys. See the Search Function Search Keys table below for RFC 2060 values for this parameter.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function retrieves the message numbers that correspond to the search criteria. You can specify a variety of search criteria, as shown in Table 11.2 in this section. These fetch criteria are defined in RFC 2060. If you specify more than one item, place the items in a parenthesized list. This function sends the SEARCH IMAP4 protocol command.

For more information, see Searching for Messages.

Table 11.2 Search Function Search Keys

Search Key	Search for messages with:
`<message set>`	Message sequence numbers corresponding to specified Message sequence number set
`ALL`	All messages in mailbox; default initial key for ANDing
`ANSWERED`	`\Answered` flag set
`BCC <string>`	Specified string in envelope's BCC field.
`BEFORE <date>`	Internal date earlier than specified date.
`BODY <string>`	Specified string in message body
`CC <string>`	Specified string in the envelope structure's CC field.
`DELETED`	`\Deleted` flag set.
`DRAFT`	`\Draft` flag set.
`FLAGGED`	`\Flagged` flag set.

Search Key	Search for messages with:
FROM <string>	Specified string in the envelope structure's FROM field.
HEADER <field-name> <string>	Header with specified field-name (as defined in [RFC-822]) and that specified string in [RFC-822] field-body.
KEYWORD <flag>	Specified keyword set.
LARGER <n>	[RFC-822] size larger than the specified number of octets.
NEW	\Recent flag set but not the \Seen flag. This is functionally equivalent to "(RECENT UNSEEN)".
NOT <search-key>	Do not match the specified search key.
OLD	No \Recent flag set; equivalent to "NOT RECENT" (as opposed to "NOT NEW").
ON <date>	Internal date within the specified date.
OR <search-key1> <search-key2>	Match either search key.
RECENT	\Recent flag set.
SEEN	\Seen flag set.
SENTBEFORE <date>	[RFC-822] Date: header earlier than specified date.
SENTON <date>	[RFC-822] Date: header within specified date.
SENTSINCE <date>	[RFC-822] Date: header within or later than specified date.
SINCE <date>	Internal date is within or later than specified date.
SMALLER <n>	[RFC-822] size smaller than specified number of octets.
SUBJECT <string>	Specified string in envelope structure's SUBJECT field.
TEXT <string>	Specified string in header or body of message.
TO <string>	Specified string in envelope structure's TO field.
UID <message set>	Unique identifiers corresponding to specified unique identifier set.
UNANSWERED	No \Answered flag set.
UNDELETED	No \Deleted flag set.
UNDRAFT	No \Draft flag set.
UNFLAGGED	No \Flagged flag set.

Search Key	Search for messages with:
UNKEYWORD <flag>	No specified keyword set.
UNSEEN	No \Seen flag set.

This function is mapped to one or more callbacks in the imap4Sink_t structure. For more information, see IMAP4 Function Callback Mapping.

Note Use imap4Tag_free to free the associated tag. §

See Also imap4Sink_t.searchStart, imap4Sink_t.search, imap4Sink_t.searchEnd, imap4Sink_t.taggedLine, imap4Sink_t.error, imap4_fetch, imap4_uidSearch, imap4Sink_t.taggedLine, imap4Sink_t.error, imap4Tag_free

[Top] [Selected State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

imap4_store

Alters data associated with one or more messages in the mailbox.

Syntax

```
#include <imap4.h>
int imap4_store(imap4Client_t* in_pimap4,
             const char* in_msgSet,
             const char* in_dataItem,
             const char* in_value,
             char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_msgSet`	Pointer to the message set.
`in_dataItem`	Pointer to the data items you can search for. See the Data Item in the Store Function Data Items table below for RFC 2060 values for this parameter.
`in_value`	Pointer to the value for the message data item. See the Value item in the Store Function Data Items table below for RFC 2060 values for this parameter.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function sends the STORE IMAP4 protocol command.

Table 11.3 Store Function Data Items

Data Item	Value	Description
`FLAGS`	`<flag list>`	Replace message flags with items in flag list. Returns the new value of the flags.
`FLAGS.SILENT`	`<flag list>`	Like `FLAGS`, but does not return a new value.
`+FLAGS`	`<flag list>`	Add the items in flag list to the message flags. Returns the new value of the flags.
`+FLAGS.SILENT`	`<flag list>`	Like `+FLAGS`, but does not return a new value.
`-FLAGS`	`<flag list>`	Remove the items in flag list from the message flags. Returns the new value of the flags.
`-FLAGS.SILENT`	`<flag list>`	Like `-FLAGS`, but does not return a new value.

You can find a link to RFC 2060 in IMAP4 RFCs.

This function is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

Note Use `imap4Tag_free` to free the associated tag. §

See Also `imap4Sink_t.taggedLine`,

imap4Sink_t.error, imap4_delete,
imap4_copy, imap4_expunge, imap4Tag_free

[Top] [Selected State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_uidCopy

Copies messages specified with a unique identifier.

Syntax

```
#include <imap4.h>
int imap4_uidCopy(imap4Client_t* in_pimap4,
             const char* in_msgSet,
             const char* in_mailbox,
             char** out_ppTagID);
```

Parameters The function has the following parameters:

in_pimap4	Pointer to the IMAP4 client.
in_msgSet	Pointer to the unique message identifiers (UIDs) to copy, for example, “1:2”
in_mailbox	Pointer to the name of target mailbox.
out_ppTagID	Pointer to pointer to the tag associated with the function. Use imap4Tag_free to free the associated memory.

Returns

- If successful, the function returns NSMAIL_OK.
- If unsuccessful, the function returns an error code. For a complete list, see “Error Definitions.”

Description Functions whose names start with UID use a unique identifier to specify messages rather than sequence numbers. This function uses the UID IMAP4 protocol command to specify that the COPY command uses unique message identifiers.

This function is mapped to one or more callbacks in the imap4Sink_t structure. For more information, see IMAP4 Function Callback Mapping.

Note Use imap4Tag_free to free the associated tag. §

See Also imap4Sink_t.taggedLine, imap4Sink_t.error, imap4_copy, imap4_expunge, imap4_store, imap4Tag_free

[Top] [Selected State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

imap4_uidFetch

Fetches messages specified with a unique identifier and retrieves the corresponding unique identifiers.

Syntax

```
#include <imap4.h>
int imap4_uidFetch ( imap4Client_t* in_pimap4,
                     const char* in_msgSet,
                     const char* in_fetchCriteria,
                     char** out_ppTagID);
```

Parameters The function has the following parameters:

in_pimap4	Pointer to the IMAP4 client.
in_msgSet	Pointer to the unique message identifiers (UIDs) to fetch, for example, "1:2".
in_fetchCriteria	Pointer to the message data item names; criteria that fetched messages must meet. See the RFC 2060 Fetch Data Items table below for RFC 2060 values for this parameter.
out_ppTagID	Pointer to pointer to the tag associated with the function. Use imap4Tag_free to free the associated memory.

Returns

- If successful, the function returns NSMAIL_OK.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function retrieves the data specified by the fetch criteria for the given message set. You can specify a variety of fetch criteria, as shown in Table 11.1 in this section. If you pass more than one item, they should be in a parenthesized list. For more information, see Fetching Message Data.

Functions whose names start with UID use a unique identifier to specify messages rather than sequence numbers. This function uses the UID IMAP4 protocol command to specify that the FETCH command uses unique message identifiers.

This function is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

Note Use `imap4Tag_free` to free the associated tag. §

See Also `imap4Sink_t.fetchStart`, `imap4Sink_t.fetchEnd`, `imap4Sink_t.fetchSize`, `imap4Sink_t.fetchData`, `imap4Sink_t.fetchFlags`, `imap4Sink_t.fetchBodyStructure`, `imap4Sink_t.fetchEnvelope`, `imap4Sink_t.fetchInternalDate`, `imap4Sink_t.fetchHeader`, `imap4Sink_t.fetchUid`, `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4_search`, `imap4_uidFetch`, `imap4Tag_free`, `imap4_fetch`

[Top] [Selected State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_uidSearch

Searches the mailbox for messages that match your search criteria and retrieve the corresponding unique identifiers.

Syntax

```
#include <imap4.h>
int imap4_uidSearch( imap4Client_t* in_pimap4,
                const char* in_criteria,
                char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_criteria`	Pointer to the search criteria consisting of one or more search keys. See the Search Function Search Keys table in the `imap4_search` function entry for RFC 2060 values for this parameter.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description You can specify a variety of search criteria, as shown in Table 11.2 in this section. If you specify more than one item, place them in a parenthesized list. For more information, see Searching for Messages.

Functions whose names start with UID use a unique identifier to specify messages rather than sequence numbers that match the search criteria. This function uses the UID imap4 protocol command to specify that the SEARCH command returns unique message identifiers.

This function is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

Note Use `imap4Tag_free` to free the associated tag. §

See Also `imap4Sink_t.searchStart`, `imap4Sink_t.search`, `imap4Sink_t.searchEnd`, `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4_search`, `imap4_uidStore`, `imap4_uidFetch`, `imap4_uidCopy`, `imap4Tag_free`

[Top] [Selected State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

imap4_uidStore

Updates message data associated with a unique identifier.

Syntax

```
#include <imap4.h>
int imap4_uidStore(imap4Client_t* in_pimap4,
              const char* in_msgSet,
              const char* in_dataItem,
              const char* in_value,
              char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_msgSet`	Pointer to the message set, specified as unique identifiers.
`in_dataItem`	Pointer to the message data item name.
`in_value`	Pointer to the value for the message data item.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description Functions whose names start with UID use a unique identifier to specify messages rather than sequence numbers. This function uses the UID IMAP4 protocol command to specify that the STORE command uses unique message identifiers.

This function is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

Note Use `imap4Tag_free` to free the associated tag. §

See Also `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4_uidSearch`, `imap4_uidStore`, `imap4_uidFetch`, `imap4_uidCopy`, `imap4_store`, `imap4Tag_free`

[Top] [Selected State Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

Extended IMAP4 Functions

Extended IMAP4 Functions provide operations involving IMAP4 extensions, such as ACL (access control list) and rights. Before these functions are available, login must take place. To find out if the server supports any extensions, use the `imap4_capability` function.

For details about extensions to IMAP4, see RFC 2086. For links to IMAP4 RFCs, see IMAP4 RFCs. Click the function name to get information about it.

Note Only Message Server 4.0 supports the ACL and Namespace extensions. §

Function	Description
`imap4_deleteACL`	Removes an identifier from the access control list for a mailbox.
`imap4_getACL`	Gets the access control list for a mailbox.
`imap4_listRights`	Finds which rights you can grant a specified user to a particular mailbox.
`imap4_myRights`	Retrieves the rights that the user has to a mailbox.
`imap4_namespace`	Finds the prefixes of namespaces used by a server for personal mailboxes, other user's mailboxes, and shared objects.
`imap4_setACL`	Changes the access control list to grant specified permissions to an identifier for a mailbox.

[Top] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

imap4_deleteACL

Remove an identifier from the access control list for a mailbox.

Syntax

```
#include <imap4.h>
int imap4_deleteACL( imap4Client_t* in_pimap4,
                     const char* in_mailbox,
```

```
                    const char* in_authID,
                    char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_mailbox`	Pointer to the mailbox.
`in_authID`	Pointer to the authentication identifier.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function revokes a user's rights to a mailbox. Given the mailbox name and an identifier, it removes the identifier/rights pair from the access control list.

This function sends the DELETEACL protocol command. It is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

Note Use `imap4Tag_free` to free the associated tag. §

See Also `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4_getACL`, `imap4_setACL`, `imap4Tag_free`

[Top] [Extended IMAP4 Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_getACL

Gets the access control list for a mailbox in an untagged ACL reply.

Syntax

```
#include <imap4.h>
int imap4_getACL( imap4Client_t* in_pimap4,
                  const char* in_mailbox,
                  char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_mailbox`	Pointer to the mailbox.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description An access control list is made up of a set of <identifier, rights> pairs. The identifier represents the authorized user. Rights include any operations that the user can perform a mailbox. These can include lookup, read, write, post, create, delete, or keep information seen or unseen. For details, see RFC 2086.

This function sends the GETACL protocol command. It is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

Note Use `imap4Tag_free` to free the associated tag. §

See Also `imap4Sink_t.aclStart`, `imap4Sink_t.aclIdentifierRight`, `imap4Sink_t.aclEnd`, `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4Tag_free`, `imap4_deleteACL`, `imap4_setACL`

[Top] [Extended IMAP4 Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_listRights

Find which rights you can grant a specified user to a particular mailbox.

Syntax

```
#include <imap4.h>
int imap4_listRights(imap4Client_t* in_pimap4,
                const char* in_mailbox,
                const char* in_authID,
                char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_mailbox`	Pointer to the mailbox.
`in_authID`	Pointer to the authentication identifier.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description Given the mailbox name and an identifier, this function tells which rights you can grant a specified user to a particular mailbox using the `imap4_setACL` function. The access control list is returned in an untagged ACL reply.

This function sends the LISTRIGHTS protocol command. It is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

Note Use `imap4Tag_free` to free the associated tag. §

See Also `imap4Sink_t.listRightsStart`, `imap4Sink_t.listRightsRequiredRights`, `imap4Sink_t.listRightsOptionalRights`, `imap4Sink_t.listRightsEnd`, `imap4_setACL`, `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4_myRights`, `imap4Tag_free`

[Top] [Extended IMAP4 Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_myRights

Retrieves the set of rights that the user has to a mailbox.

Syntax

```
#include <imap4.h>
int imap4_myRights(imap4Client_t* in_pimap4,
                   const char* in_mailbox,
```

```
                    char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_mailbox`	Pointer to the mailbox.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description Rights include any operations that the user can perform a mailbox. These can include lookup, read, write, post, create, delete, or keep information seen or unseen. For details, see RFC 2086.

This function sends the MYRIGHTS protocol command. It is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

Note Use `imap4Tag_free` to free the associated tag. §

See Also `imap4Sink_t.myRights`, `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4_listRights`, `imap4Tag_free`

[Top] [Extended IMAP4 Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_namespace

Finds the prefixes of namespaces used by a server for personal mailboxes, other user's mailboxes, and shared mailboxes.

Syntax

```
#include <imap4.h>
int imap4_nameSpace( imap4Client_t* in_pimap4,
                     char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description Different types of mailbox stores, such as newsgroups or corporate mailboxes, have their own namespace identifiers. The namespace is identified by prefixing # and the store name at the beginning of the mailbox address. For example, `#news` can identify an address as a newsgroup mailbox name.

This function issues the NAMESPACE command. It is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

Note Use `imap4Tag_free` to free the associated tag. §

See Also `imap4Sink_t.nameSpaceStart`, `imap4Sink_t.nameSpacePersonal`, `imap4Sink_t.nameSpaceOtherUsers`, `imap4Sink_t.nameSpaceShared`, `imap4Sink_t.nameSpaceEnd`, `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4Tag_free`

[Top] [Extended IMAP4 Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

---

imap4_setACL

Changes the access control list to grant specified permissions to an identifier for a mailbox.

Syntax

```
#include <imap4.h>
int imap4_setACL( imap4Client_t* in_pimap4,
                  const char* in_mailbox,
```

```
const char* in_authID,
const char* in_accessRight,
char** out_ppTagID);
```

Parameters The function has the following parameters:

`in_pimap4`	Pointer to the IMAP4 client.
`in_mailbox`	Pointer to the mailbox.
`in_authID`	Pointer to the authentication identifier.
`in_accessRight`	Pointer to access rights to grant.
`out_ppTagID`	Pointer to pointer to the tag associated with the function. Use `imap4Tag_free` to free the associated memory.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description Functions that have ACL in their names are able to manipulate the access control list (ACL), as described in RFC 2086. You can find out if the server supports ACL by issuing the `imap4_capability` command.

The identifier specified in the `in_authID` parameter is granted rights as specified in the `in_accessRight` parameter. Rights include any operations that the user can perform a mailbox. These can include lookup, read, write, post, create, delete, or keep information seen or unseen. For details, see RFC 2086.

This function sends the SETACL protocol command. It is mapped to one or more callbacks in the `imap4Sink_t` structure. For more information, see IMAP4 Function Callback Mapping.

Note Use `imap4Tag_free` to free the associated tag. §

See Also `imap4Sink_t.aclStart`, `imap4Sink_t.taggedLine`, `imap4Sink_t.error`, `imap4_deleteACL`, `imap4_getACL`, `imap4_myRights`, `imap4Tag_free`

[Top] [Extended IMAP4 Functions] [IMAP4 Functions by Functional Group] [IMAP4 Functions by Task]

IMAP4 Structures

This section defines IMAP4 data structures, listed in alphabetical order.

```
imap4Client_t        imap4Sink_t
```

[Top]

---

imap4Client_t

Contains a reference to all major objects within the IMAP SDK, including data members.

Syntax

```
#include <imap4.h>
typedef struct IMAP4 imap4Client_t;
```

Description The client uses this structure to communicate with an IMAP4 server. The developer uses this structure to perform IMAP4 API operations. All data contained within the client structure is used internally by the API. For more information, see Creating a Client.

See Also `imap4Sink_t`

[Top] [IMAP4 Structures]

---

imap4Sink_t

Represents the response sink for all IMAP4 commands.

Syntax

```
#include <imap4.h>
typedef struct IMAP4Sink
{
void *pOpaqueData;
void (*taggedLine)(
```

```
        imap4SinkPtr_t in_pimap4Sink,
        char* in_tag,
        const char* in_status,
        const char* in_reason);
void (*error)(
        imap4SinkPtr_t in_pimap4Sink,
        const char* in_tag,
        const char* in_status,
        const char* in_reason);
void (*ok)(
        imap4SinkPtr_t in_pimap4Sink,
        const char* in_responseCode,
        const char* in_information);
void (*rawResponse)(
        imap4SinkPtr_t in_pimap4Sink,
        const char* in_data);

/*Fetch Response*/

void* (*fetchStart)(
        imap4SinkPtr_t in_pimap4Sink,
        int in_msg);
void (*fetchEnd)(
        imap4SinkPtr_t in_pimap4Sink);
void (*fetchSize)(
        imap4SinkPtr_t in_pimap4Sink,
        int in_size);
void (*fetchData)(
        imap4SinkPtr_t in_pimap4Sink,
        const char* in_data,
        int in_bytesRead,
        int in_totalBytes);
void (*fetchFlags)(
        imap4SinkPtr_t in_pimap4Sink,
        const char* in_flags);
void (*fetchBodyStructure)(
        imap4SinkPtr_t in_pimap4Sink,
        const char* in_bodyStructure);
void (*fetchEnvelope)(
        imap4SinkPtr_t in_pimap4Sink,
        const char** in_value,
        int in_valueLength);
void (*fetchInternalDate)(
        imap4SinkPtr_t in_pimap4Sink,
        const char* in_internalDate);
void (*fetchHeader)(
```

```
            imap4SinkPtr_t in_pimap4Sink,
            const char* in_field,
            const char* in_value);
void (*fetchUid)(imap4SinkPtr_t in_pimap4Sink,
            int in_uid);

/*Lsub Response*/

void (*lsub)(
            imap4SinkPtr_t in_pimap4Sink,
            const char* in_attribute,
            const char* in_delimeter,
            const char* in_name);

/*List Response*/

void (*list)(
            imap4SinkPtr_t in_pimap4Sink,
            const char* in_attribute,
            const char* in_delimeter,
            const char* in_name);

/*Search Response*/

void* (*searchStart)(
            imap4SinkPtr_t in_pimap4Sink);
void (*search)(
            imap4SinkPtr_t in_pimap4Sink,
            int in_message);
void (*searchEnd)(
            imap4SinkPtr_t in_pimap4Sink);

/*Status Response*/

void (*statusMessages)(
            imap4SinkPtr_t in_pimap4Sink,
            int in_messages);
void (*statusRecent)(
            imap4SinkPtr_t in_pimap4Sink,
            int in_recent);
void (*statusUidnext)(
            imap4SinkPtr_t in_pimap4Sink,
            int in_uidNext);
void (*statusUidvalidity)(
            imap4SinkPtr_t in_pimap4Sink,
            int in_uidValidity);
void (*statusUnseen)(
```

```
        imap4SinkPtr_t in_pimap4Sink,
        int in_unSeen);

/*Capability Response*/

void (*capability)(
        imap4SinkPtr_t in_pimap4Sink,
        const char* in_listing);

/*Exists Response*/

void (*exists)(
        imap4SinkPtr_t in_pimap4Sink,
        int in_messages);

/*Expunge Response*/

void (*expunge)(
        imap4SinkPtr_t in_pimap4Sink,
        int in_message);

/*Recent Response*/

void (*recent)(
        imap4SinkPtr_t in_pimap4Sink,
        int in_messages);

/*Flags Response*/

void (*flags)(
        imap4SinkPtr_t in_pimap4Sink,
        const char* in_flags);

/*Bye Response*/

void (*bye)(
        imap4SinkPtr_t in_pimap4Sink,
        const char* in_reason);

/*Namespace Response*/

void* (*nameSpaceStart)(
        imap4SinkPtr_t in_pimap4Sink);
void (*nameSpacePersonal)(
        imap4SinkPtr_t in_pimap4Sink,
        const char* in_personal);
void (*nameSpaceOtherUsers)(
        imap4SinkPtr_t in_pimap4Sink,
```

```
        const char* in_otherUsers);
void (*nameSpaceShared)(
        imap4SinkPtr_t in_pimap4Sink,
        const char* in_shared);
void (*nameSpaceEnd)(
        imap4SinkPtr_t in_pimap4Sink);

/*ACL Responses*/

void* (*aclStart)(
        imap4SinkPtr_t in_pimap4Sink,
        const char* in_mailbox);
void (*aclIdentifierRight)(
        imap4SinkPtr_t in_pimap4Sink,
        const char* in_identifier,
        const char* in_rights);
void (*aclEnd)(imap4SinkPtr_t in_pimap4Sink);

/*LISTRIGHTS Responses*/

void* (*listRightsStart)(
        imap4SinkPtr_t in_pimap4Sink,
        const char* in_mailbox,
        const char* in_identifier);
void (*listRightsRequiredRights)(
        imap4SinkPtr_t in_pimap4Sink,
        const char* in_requiredRights);
void (*listRightsOptionalRights)(
        imap4SinkPtr_t in_pimap4Sink,
        const char* in_optionalRights);
void (*listRightsEnd)(
        imap4SinkPtr_t in_pimap4Sink);

/*MYRIGHTS Responses*/

void (*myRights)(
        imap4SinkPtr_t in_pimap4Sink,
        const char* in_mailbox,
        const char* in_rights);
}imap4Sink_t;
```

Fields The structure has the following fields:

`void *pOpaqueData;`	Opaque user data.

`void (*taggedLine)(`
`  imap4SinkPtr_t in_pimap4Sink,`
`  char* in_tag,`
`  const char* in_status,`
`  const char* in_reason);`
	Tagged line. Indicates the end of response(s) for successful commands. Parameters: the response sink to use, the tag associated with the command, response code, descriptive text.

`void (*error)(`
`  imap4SinkPtr_t in_pimap4Sink,`
`  const char* in_tag,`
`  const char* in_status,`
`  const char* in_reason);`
	Notification of an error. Indicates the response for unsuccessful commands. Parameters: the response sink to use, the tag associated with the command, response code, descriptive text.

`void (*ok)(`
`  imap4SinkPtr_t in_pimap4Sink,`
`  const char* in_responseCode,`
`  const char* in_information);`
	Unsolicited OK response. Parameters: the response sink to use, response code, additional text.

`void (*no)(`
`  imap4SinkPtr_t in_pimap4Sink,`
`  const char* in_responseCode,`
`  const char* in_information);`
	NO response. Indicates an operational error message from the server. Parameters: the response sink to use, response code, additional text.

`void (*bad)(`
`  imap4SinkPtr_t in_pimap4Sink,`
`  const char* in_responseCode,`
`  const char* in_information);`
	BAD response: Indicates an error message for the server. Parameters: the response sink to use, response code, additional text.

`xvoid (*rawResponse)(`
`  imap4SinkPtr_t in_pimap4Sink,`
`  const char* in_data);`
	Raw (unparsed) data is pushed into here. Parameters: the response sink to use, raw data.

`void* (*fetchStart)(`
`  imap4SinkPtr_t in_pimap4Sink,`
`  int in_msg);`
	Indicates the beginning of a fetch response. Parameters: the response sink to use, the message number.

`void (*fetchEnd)(` `  imap4SinkPtr_t in_pimap4Sink);`	Indicates that the data for this message is completely fetched. Parameter: the response sink to use.

`void (*fetchSize)(`
`  imap4SinkPtr_t in_pimap4Sink,`
`  int in_size);`
	Size of the message. Parameters: the response sink to use, size of message.

`void (*fetchData)(`
`  imap4SinkPtr_t in_pimap4Sink,`
`  const char* in_data,`
`  int in_bytesRead,`
`  int in_totalBytes);`
	Fetches data associated with specific data items. Parameters: the response sink to use, message data, number of bytes read, total number of bytes to be returned.

```
void (*fetchFlags)(
   imap4SinkPtr_t in_pimap4Sink,
   const char* in_flags);
```

Fetches the value of the flags. Parameters: the response sink to use, flags on the message.

```
void (*fetchBodyStructure)(
   imap4SinkPtr_t in_pimap4Sink,
   const char* in_bodyStructure);
```

Fetches the value of the body structure. Parameters: the response sink to use, message body structure.

```
void (*fetchEnvelope)(
   imap4SinkPtr_t in_pimap4Sink,
   const char** in_value,
   int in_valueLength);
```

Fetches the value of the envelope. Parameters: the response sink to use, an array of the envelope fields, length of the `in_value` array.

```
void (*fetchInternalDate)(
   imap4SinkPtr_t in_pimap4Sink,
   const char* in_internalDate);
```

Fetches the value of the internal date. Parameters: the response sink to use, internal date.

```
void (*fetchHeader)(
   imap4SinkPtr_t in_pimap4Sink,
   const char* in_field,
   const char* in_value);
```

Fetches the value of the header. Parameters: the response sink to use, followed by name and value parts of header name:value pair.

```
void (*fetchUid)(
   imap4SinkPtr_t in_pimap4Sink,
   int in_uid);
```

Fetches the value of the unique ID of the message. Parameters: the response sink to use, unique id of message.

```
void (*lsub)(
   imap4SinkPtr_t in_pimap4Sink,
   const char* in_attribute,
   const char* in_delimeter,
   const char* in_name);
```

Lists subscribed or active mailboxes that match the search criteria. Parameters: the response sink to use, attribute message, delimiter of message, mailbox name.

```
void (*list)(
   imap4SinkPtr_t in_pimap4Sink,
   const char* in_attribute,
   const char* in_delimeter,
   const char* in_name);
```

Lists mailboxes that match the search criteria. Parameters: the response sink to use, attribute message, delimiter of message, mailbox name.

```
void* (*searchStart)(
   imap4SinkPtr_t in_pimap4Sink);
```

Indicates the start of the retrieval of messages numbers that match the search criteria. Parameter: response sink to use.

```
void (*search)(
   imap4SinkPtr_t in_pimap4Sink,
   int in_message);
```

Gets the messages that match the search criteria. Parameters: the response sink to use, message to search for.

```
void (*searchEnd)(
   imap4SinkPtr_t in_pimap4Sink);
```

Indicates the end of the retrieval of message numbers that match the search criteria. Parameter: the response sink to use.

```
void (*statusMessages)(
   imap4SinkPtr_t in_pimap4Sink,
   int in_messages);
```

Total number of messages. Parameters: the response sink to use, messages to count.

```
void (*statusRecent)(
   imap4SinkPtr_t in_pimap4Sink,
   int in_recent);
```

Number of messages with the recent flag set. Parameter: the response sink to use.

```
void (*statusUidnext)(
   imap4SinkPtr_t in_pimap4Sink,
   int in_uidNext);
```

UID value to assign to a new message in the mailbox. Parameters: the response sink to use, unique id of the next message.

```
void (*statusUidvalidity)(
   imap4SinkPtr_t in_pimap4Sink,
   int in_uidValidity);
```

UID validity value of the mailbox. Parameters: the response sink to use, unique id to validate.

```
void (*statusUnseen)(
   imap4SinkPtr_t in_pimap4Sink,
   int in_unSeen);
```

Number of messages without the `\Seen` flag set. Parameters: the response sink to use, unique id of message flagged `\Unseen`.

```
void (*capability)(
   imap4SinkPtr_t in_pimap4Sink,
   const char* in_listing);
```

Listing of capabilities supported by the server. Parameters: the response sink to use, list of server extensions.

```
void (*exists)(
   imap4SinkPtr_t in_pimap4Sink,
   int in_messages);
```

Total number of messages in selected mailbox. Parameters: the response sink to use, number of messages in the mailbox.

```
void (*expunge)(
   imap4SinkPtr_t in_pimap4Sink,
   int in_message);
```

Expunges the specified message. Parameters: the response sink to use, number of messages.

```
void (*recent)(
   imap4SinkPtr_t in_pimap4Sink,
   int in_messages);
```

Total number of messages with the `\Recent` flag set. Parameters: the response sink to use, number of messages.

```
void (*flags)(
   imap4SinkPtr_t in_pimap4Sink,
   const char* in_flags);
```

Applicable flags for the selected mailbox. Parameters: the response sink to use, list of flags.

```
void (*bye)(
   imap4SinkPtr_t in_pimap4Sink,
   const char* in_reason);
```

Reason the connection was closed. Parameters: the response sink to use.

```
void* (*nameSpaceStart)(
   imap4SinkPtr_t in_pimap4Sink);
```

Indicates the beginning of a namespace response. Parameter: the response sink to use.

```
void (*nameSpacePersonal)(
   imap4SinkPtr_t in_pimap4Sink,
   const char* in_personal);
```

Personal namespace. Parameters: the response sink to use, the user.

`void (*nameSpaceOtherUsers)( imap4SinkPtr_t in_pimap4Sink, const char* in_otherUsers);`	Other user's namespace. Parameters: the response sink to use, list of other users.
`void (*nameSpaceShared)( imap4SinkPtr_t in_pimap4Sink, const char* in_shared);`	Shared namespace. Parameters: the response sink to use, list of users with rights to shared mailbox.
`void (*nameSpaceEnd)( imap4SinkPtr_t in_pimap4Sink);`	Signals completion of fetch of data for the particular namespace. Parameters: the response sink to use.
`void* (*aclStart)( imap4SinkPtr_t in_pimap4Sink, const char* in_mailbox);`	Mailbox name for which the ACL applies. Parameters: the response sink to use, identifier of the mailbox.
`void (*aclIdentifierRight)( imap4SinkPtr_t in_pimap4Sink, const char* in_identifier, const char* in_rights);`	Identifier rights pairs that apply to mailbox specified in `aclStart`. Parameters: response sink to use, identifier of the user, list of rights to the mailbox.
`void (*aclEnd)( imap4SinkPtr_t in_pimap4Sink);`	End of an ACL response. Parameters: the response sink to use.
`void* (*listRightsStart)( imap4SinkPtr_t in_pimap4Sink, const char* in_mailbox, const char* in_identifier);`	Start of response to the LISTRIGHTS command. Parameters: response sink to use, identifier for the mailbox, identifier of the user.
`void (*listRightsRequiredRights)( imap4SinkPtr_t in_pimap4Sink, const char* in_requiredRights);`	Required rights for the identifier defined in `listRightsStart`. Parameters: response sink to use, list of required rights to the mailbox.
`void (*listRightsOptionalRights)( imap4SinkPtr_t in_pimap4Sink, const char* in_optionalRights);`	Optional rights for the identifier defined in `listRightsStart`. Parameters: response sink to use, list of optional rights to the mailbox.
`void (*listRightsEnd)( imap4SinkPtr_t in_pimap4Sink);`	End of a LISTRIGHTS response. Parameters: response sink to use.
`void (*myRights)( imap4SinkPtr_t in_pimap4Sink, const char* in_mailbox, const char* in_rights);`	Set of rights that the user has to the mailbox. Parameters: response sink to use, identifier for the mailbox, list of rights to the mailbox.

Description The IMAP4 response sink structure is made up of function pointers and opaque data. You must define the functions yourself, and you must point these pointers to your functions if you want to receive the related server responses. For more information, see SDK Response Sinks for C and Creating a Response Sink.

The function pointers serve as callbacks for many IMAP4 functions. See IMAP4 Function Callback Mapping.

Note Use `imap4Tag_free` to free tags generated by IMAP4 functions. §

See Also `imap4_initialize, imap4Sink_free, imap4Tag_free`

[Top] [IMAP4 Structures]

Chapter

12

POP3 C API Reference

This chapter describes the functions and structures of the C language API for the POP3 (Post Office Protocol) protocol of the Messaging Access SDK.

- POP3 Functions
- POP3 Structures

You'll find links to each function and structure in this introduction. Each reference entry gives the name of the function or structure, its header file, its syntax, and its parameters.

All C interface definitions are found in the `pop3.h` file.

[Top]

POP3 Functions

This table lists POP3 functions alphabetically by name, with descriptions. Click the function name to get information about it.

Function	Description
pop3_connect	Connects to server.
pop3_delete	Marks a message for deletion on the server.
pop3_disconnect	Closes the socket connection with the server.
pop3_free	Frees the pop3Client_t structure.
pop3_get_option	Gets IO models.
pop3_initialize	Initializes and allocates the pop3Client_t structure and sets the response sink.
pop3_list	Lists all messages.
pop3_listA	Lists the specified message.
pop3_noop	Contacts the server, which sends a response indicating it is still present.
pop3_pass	Identifies user password; on success, enters Transaction state.
pop3_processResponses	Processes the server responses for API commands.
pop3_quit	Closes the connection with the server.
pop3_reset	Undeletes any messages marked as deleted.
pop3_retrieve	Retrieves message with specified number.
pop3_sendCommand	Send an unsupported command to the server.
pop3_setChunkSize	Sets the size of the message data chunk passed to the user.
pop3_set_option	Sets IO models.
pop3_setResponseSink	Sets a new response sink.
pop3_setTimeout	Sets the amount of time allowed to wait before returning control to the user.
pop3_stat	Get the status of the mail drop.

Function	Description
pop3_top	Retrieves the headers plus the specified number of lines from the message.
pop3_uidList	Returns the message numbers and the corresponding unique ids of the message in the maildrop.
pop3_uidListA	Returns the specified message number and the corresponding unique id of the message.
pop3_user	Enters a user name.
pop3_xAuthList	Returns a list of authenticated users.
pop3_uidListA	Returns an authenticated user for a given message.
pop3_xSender	Gets the email address of the sender of the specified message.
pop3Sink_free	Frees the POP3 response sink and its data members.
pop3Sink_initialize	Initializes and allocates the pop3SinkPtr_t structure.

[Top] [POP3 Functions] [POP3 Functions by Task]

pop3_connect

Connects to the server.

Syntax

```
#include <pop3.h>
int pop3_connect( pop3Client_t * in_pPOP3,
                  const char * in_server,
                  unsigned short in_port );
```

Parameters The function has the following parameters:

in_pPOP3	Pointer to the POP3 client.
in_server	Pointer to the name of the server to which to connect.
in_port	Server port to which to connect.

Returns

- If successful, the function returns NSMAIL_OK.

- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function connects the POP3 client to the specified server at the port you indicate. For more information, see Connecting to a Server.

This function is mapped to one or more callbacks in the `pop3Sink_t` structure. For more information, see POP3 Function Callback Mapping.

See Also `pop3Sink_t.connect`, `pop3Sink_t.error`, `pop3_disconnect`, `pop3Client_t`

[Top] [POP3 Functions] [POP3 Functions by Task]

---

pop3_delete

Marks a message for deletion on the server.

Syntax

```
#include <pop3.h>
int pop3_delete( pop3Client_t * in_pPOP3,
            int in_messageNumber );
```

Parameters The function has the following parameters:

`in_pPOP3`	Pointer to the POP3 client.
`in_messageNumber`	Message number.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function marks the message for deletion; it is actually deleted when the `pop3_quit` function is called. For more information, see Ending the Session.

This function sends the DELE [arg] POP3 protocol command. It is mapped to one or more callbacks in the `pop3Sink_t` structure. For more information, see POP3 Function Callback Mapping.

See Also `pop3Sink_t.dele`, `pop3Sink_t.error`, `pop3_quit`, `pop3Client_t`

[Top] [POP3 Functions] [POP3 Functions by Task]

pop3_disconnect

Closes the socket connection with the server.

Syntax

```
#include <pop3.h>
int pop3_disconnect ( pop3Client_t * in_pPOP3 );
```

Parameters The function has the following parameter:

`in_pPOP3` Pointer to the POP3 client you want to disconnect.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function closes the socket connection. It could be used to implement a Cancel button.

Note Do not call the `processResponses` method after this function. Operations for which you do not call `processResponses`: disconnecting, the set functions, initializing and freeing the client and sink. §

See Also `pop3_connect`

[Top] [POP3 Functions] [POP3 Functions by Task]

pop3_free

Frees the POP3 client structure.

Syntax

```
#include <pop3.h>
void pop3_free( pop3Client_t ** in_ppPOP3 );
```

Parameters The function has the following parameter:

`in_pPOP3`	Pointer to pointer to the client structure.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function frees the `pop3Client_t` structure. When the function returns, the client structure is set to `null`.

See Also `pop3_initialize, pop3Sink_free`

[Top] [POP3 Functions] [POP3 Functions by Task]

---

pop3_get_option

Gets IO models.

Syntax

```
#include <pop3.h>
int pop3_get_option( pop3Client_t * in_pPOP3,
                int in_option,
                void * in_pOptionData );
```

Parameters The function has the following parameters:

`in_pPOP3`	Pointer to the POP3 client.
`in_option`	Pointer to option you want to set. See `nsmail_io_fns_t`. For the option that is currently supported, see "Option Definition."
`in_pOptionData`	Pointer to option data set with `pop3_set_option`.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description This function gets the option set with `pop3_set_option`. Do not call `pop3_processResponses` after this function.

See Also `pop3_set_option`

[Top] [POP3 Functions] [POP3 Functions by Task]

---

pop3_initialize

Initializes and allocates the `pop3Client_t` structure and sets the response sink.

Syntax

```
#include <pop3.h>
int pop3_initialize( pop3Client_t ** out_ppPOP3,
                     pop3SinkPtr_t in_pPOP3Sink );
```

Parameters The function has the following parameters:

`out_ppPOP3`	Pointer to pointer to the POP3 client.
`in_pPOP3Sink`	Response sink to use.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description For more information, see Creating a Client.

See Also `pop3Client_t`

[Top] [POP3 Functions] [POP3 Functions by Task]

---

pop3_list

Lists all messages.

Syntax

```
#include <pop3.h>
int pop3_list( pop3Client_t * in_pPOP3 );
```

Parameters The function has the following parameters:

`in_pPOP3`	Pointer to the POP3 client.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description POP3 includes two forms of the list function. This form goes through all the messages in the mailbox and generates a list of messages. The other form takes the message number of the message to list as an argument. For more information, see Listing Messages.

This function sends the LIST POP3 protocol command. It is mapped to one or more callbacks in the `pop3Sink_t` structure. For more information, see POP3 Function Callback Mapping.

See Also `pop3Sink_t.listStart`, `pop3Sink_t.list`, `pop3Sink_t.listComplete`, `pop3Sink_t.error`, `pop3_listA`, `pop3Client_t`

[Top] [POP3 Functions] [POP3 Functions by Task]

---

pop3_listA

Lists the specified message.

Syntax

```
#include <pop3.h>
int pop3_listA( pop3Client_t * in_pPOP3,
            int in_messageNumber );
```

Parameters The function has the following parameters:

`in_pPOP3`	Pointer to the POP3 client.
`in_messageNumber`	Message number of the message to list.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description POP3 includes two forms of the list function. This form takes the message number of the message to list as an argument. The other form goes through all the messages in the mailbox and generates a list of messages. For more information, see Listing Messages.

This function sends the LIST [arg] POP3 protocol command. It is mapped to one or more callbacks in the `pop3Sink_t` structure. For more information, see POP3 Function Callback Mapping.

See Also `pop3Sink_t.listStart`, `pop3Sink_t.list`, `pop3Sink_t.listComplete`, `pop3Sink_t.error`, `pop3_list`

[Top] [POP3 Functions] [POP3 Functions by Task]

---

pop3_noop

Contacts the server, which sends a response indicating it is still present.

Syntax
```
#include <pop3.h>
int pop3_noop( pop3Client_t * in_pPOP3 );
```

Parameters The function has the following parameters:

`in_pPOP3`	Pointer to the POP3 client.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function sends the NOOP POP3 protocol command.

If you send any command, the server responds with an "still here" response. Sending the `pop3_noop` function does nothing except force this server response. You can use this function to maintain server connection, perhaps issuing it at timed intervals to make sure that the server is still active.

The `pop3_noop` function resets the autologout timer inside the server and may change the number of messages in a mailbox (due to new mail). Also, using this function may change or update the status of messages in mailboxes.

Your application may not need this function. For example, if the application does something and then disconnects at once, there is no need to make sure that the server is still connected.

This function is mapped to one or more callbacks in the `pop3Sink_t` structure. For more information, see POP3 Function Callback Mapping.

See Also `pop3Sink_t.noop`, `pop3Sink_t.error`, `pop3_connect`, `pop3Client_t`

[Top] [POP3 Functions] [POP3 Functions by Task]

---

pop3_pass

Identifies user password; on success, enters Transaction state.

Syntax

```
#include <pop3.h>
int pop3_pass( pop3Client_t * in_pPOP3,
               const char * in_password );
```

Parameters The function has the following parameters:

`in_pPOP3`	Pointer to the POP3 client.
`in_password`	Pointer to the user password.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function sends the PASS POP3 protocol command. For more information, see Logging In.

This function is mapped to one or more callbacks in the `pop3Sink_t` structure. For more information, see POP3 Function Callback Mapping.

See Also `pop3Sink_t.pass`, `pop3Sink_t.error`, `pop3_user`

[Top] [POP3 Functions] [POP3 Functions by Task]

pop3_processResponses

Processes the server responses for API commands.

Syntax

```
#include <pop3.h>
int pop3_processResponses( pop3Client_t * in_pPOP3);
```

Parameters The function has the following parameters:

`in_pPOP3` Pointer to the POP3 client.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If a time-out occurs, the function returns `NSMAIL_ERR_TIMEOUT`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function processes the server responses for API commands. It invokes the callback functions provided by the user for all responses that are available at the time of execution.

If a time-out occurs, the user can continue by calling `pop3_processResponses` again.

Do not call `pop3_processResponses` after these operations: disconnecting, the set functions, initializing and freeing the client and sink.

See Also `pop3Sink_t`

[Top] [POP3 Functions] [POP3 Functions by Task]

pop3_quit

Closes the connection with the server.

Syntax

```
#include <pop3.h>
int pop3_quit( pop3Client_t * in_pPOP3 );
```

Parameters The function has the following parameters:

`in_pPOP3`	Pointer to the POP3 client.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function ends the session. It expunges deleted messages and logs the user out from the POP3 server. For more information, see Ending the Session.

If issued in the Authentication state, server closes connections. If issued in the Transaction state, the server enters the Update state and deletes marked messages, then quits.

This function sends the QUIT POP3 protocol command. It is mapped to one or more callbacks in the `pop3Sink_t` structure. For more information, see POP3 Function Callback Mapping.

See Also `pop3Sink_t.quit`, `pop3Sink_t.error`, `pop3_delete`, `pop3Client_t`

[Top] [POP3 Functions] [POP3 Functions by Task]

---

pop3_reset

Undeletes any messages marked as deleted.

Syntax

```
#include <pop3.h>
int pop3_reset( pop3Client_t * in_pPOP3 );
```

Parameters The function has the following parameters:

`in_pPOP3`	Pointer to the POP3 client.

Returns

- If successful, the function returns `NSMAIL_OK`.

- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function removes any delete flags.

This function sends the RSET POP3 protocol command. It is mapped to one or more callbacks in the `pop3Sink_t` structure. For more information, see POP3 Function Callback Mapping.

See Also `pop3Sink_t.reset`, `pop3Sink_t.error`, `pop3Client_t`

[Top] [POP3 Functions] [POP3 Functions by Task]

---

pop3_retrieve

Retrieves message with specified number.

Syntax

```
#include <pop3.h>
int pop3_retrieve( pop3Client_t * in_pPOP3,
                   int in_messageNumber );
```

Parameters The function has the following parameters:

`in_pPOP3`	Pointer to the POP3 client.
`in_messageNumber`	Number of message to retrieve.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function retrieves messages in the form of data chunks, which have a default size of 1 K. The function issues a RETR IMAP4 protocol command. For more information, see Retrieving a Message.

This function is mapped to one or more callbacks in the `pop3Sink_t` structure. For more information, see POP3 Function Callback Mapping.

See Also `pop3Sink_t.retrieveStart`, `pop3Sink_t.retrieve`, `pop3Sink_t.retrieveComplete`, `pop3Sink_t.error`, `pop3_setChunkSize`

[Top] [POP3 Functions] [POP3 Functions by Task]

pop3_sendCommand

Sends a POP3 command that is not otherwise supported by the SDK.

Syntax

```
#include <pop3.h>
int pop3_sendCommand( pop3Client_t * in_pPOP3,
                const char * in_command,
                boolean in_multiLine );
```

Parameters The function has the following parameters:

`in_pPOP3`	Pointer to the POP3 client.
`in_command`	Pointer to the command to send.
`in_multiLine`	Whether the command is multi-line. • `true`: multi-line. • `false`: single line.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description You can use this function to extend the protocol to meet your application needs. Using this function, you can extend the functionality of the SDK to meet your needs by adding functions or passing different parameters to a function.

This function is mapped to one or more callbacks in the `pop3Sink_t` structure. For more information, see POP3 Function Callback Mapping.

See Also `pop3Sink_t.sendCommandStart`, `pop3Sink_t.sendCommand`, `pop3Sink_t.sendCommandComplete`

[Top] [POP3 Functions] [POP3 Functions by Task]

pop3_setChunkSize

Sets the size of the message data chunk passed to the user.

Syntax

```
#include <pop3.h>
int pop3_setChunkSize( pop3Client_t * in_pPOP3,
                       int in_chunkSize );
```

Parameters The function has the following parameters:

`in_pPOP3`	Pointer to the POP3 client.
`in_chunkSize`	Size of chunk to set.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description

Note Do not call the `processResponses` method after this method. §

See Also `pop3_retrieve`

[Top] [POP3 Functions] [POP3 Functions by Task]

---

pop3_set_option

Sets IO models.

Syntax

```
#include <pop3.h>
int pop3_set_option( pop3Client_t * in_pPOP3,
                     int in_option,
                     void * in_pOptionData );
```

Parameters The function has the following parameters:

`in_pPOP3`	Pointer to the POP3 client.
`in_option`	Pointer to option you want to set. See `nsmail_io_fns_t`. For the option that is currently supported, see "Option Definition."
`in_pOptionData`	Pointer to the option data you want to set.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error code. For a complete list, see "Error Definitions."

Description To inquire on options set with this function, use `pop3_get_option`. Do not call `pop3_processResponses` after this function.

See Also `pop3_get_option`

[Top] [POP3 Functions] [POP3 Functions by Task]

---

pop3_setResponseSink

Sets a new response sink.

Syntax

```
#include <pop3.h>
int pop3_setResponseSink ( pop3Client_t * in_pPOP3,
                           pop3SinkPtr_t in_pPOP3Sink );
```

Parameters The function has the following parameters:

`in_pPOP3`	Pointer to the POP3 client.
`in_pPOP3Sink`	Response sink to set.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function overrides the response sink passed into the initialized function.

Note Do not call `pop3_processResponses` after this function. §

See Also pop3Sink_t

[Top] [POP3 Functions] [POP3 Functions by Task]

---

pop3_setTimeout

Sets the amount of time allowed to wait before returning control to the user.

Syntax

```
#include <pop3.h>
int pop3_setTimeout( pop3Client_t * in_pPOP3,
                     double in_timeout );
```

Parameters The function has the following parameters:

in_pPOP3	Pointer to the POP3 client.
in_timeout	Time-out period to set in seconds. Values: • -1 = infinite time-out (default) • 0 = no waiting • >0 = length of time-out period

Returns

- If successful, the function returns NSMAIL_OK.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function sets the amount of time, in seconds, allowed to wait before returning control to the user.

Note Do not call pop3_processResponses after this function. §

See Also pop3_processResponses, pop3_setChunkSize, pop3Client_t

[Top] [POP3 Functions] [POP3 Functions by Task]

---

pop3Sink_free

Frees the POP3 response sink and its data members.

Syntax

```
#include <pop3.h>
void pop3Sink_free( pop3Sink_t ** in_ppPOP3Sink );
```

Parameters The function has the following parameters:

out_ppPOP3Sink	Pointer to a pointer to the client structure associated with the response sink to free.

Returns

- If successful, the function returns NSMAIL_OK.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function frees the pop3Sink_t structure. When the function returns, the sink structure is set to null. The user must free any pointers to opaque data.

Note Do not call pop3_processResponses after this function. §

See Also pop3_quit, pop3_free, pop3Sink_initialize

[Top] [POP3 Functions] [POP3 Functions by Task]

---

pop3Sink_initialize

Initializes and allocates the pop3Sink_t structure.

Syntax

```
#include <smtp.h>
int pop3Sink_initialize( pop3Sink_t ** out_ppPOP3Sink );
```

Parameters The function has the following parameters:

out_ppPOP3Sink	Pointer to pointer to the response sink to initialize.

Returns

- If successful, the function returns NSMAIL_OK.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description For more information, see Creating a Response Sink and SDK Response Sinks for C.

Note Do not call the `processResponses` method after this method. §

See Also `pop3_connect, pop3_noop, pop3Sink_free, pop3_initialize`

[Top] [POP3 Functions] [POP3 Functions by Task]

---

pop3_stat

Gets the status of the mail drop.

Syntax

```
#include <pop3.h>
int pop3_stat( pop3Client_t * in_pPOP3 );
```

Parameters The function has the following parameters:

`in_pPOP3`	Pointer to the POP3 client.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description The client calls this function to get the number of messages in and octet size of the mail drop. The function issues a STAT IMAP4 protocol command. For more information, see Getting Message Count.

This function is mapped to one or more callbacks in the `pop3Sink_t` structure. For more information, see POP3 Function Callback Mapping.

See Also `pop3Sink_t.stat`, `pop3Sink_t.error`

[Top] [POP3 Functions] [POP3 Functions by Task]

---

pop3_top

Retrieves the headers plus the specified number of lines from the body.

Syntax

```
#include <pop3.h>
int pop3_top ( pop3Client_t * in_pPOP3,
               int in_messageNumber,
               int in_lines );
```

Parameters The function has the following parameters:

`in_pPOP3`	Pointer to the POP3 client.
`in_messageNumber`	Number of message to retrieve.
`in_lines`	Number of lines of the body to retrieve.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function returns the header and specified number of lines from the message. It issues a TOP [arg1] [arg2]' command. For more information, see Retrieving Message Headers.

This function is mapped to one or more callbacks in the `pop3Sink_t` structure. For more information, see POP3 Function Callback Mapping.

See Also `pop3Sink_t.topStart`, `pop3Sink_t.top`, `pop3Sink_t.topComplete`, `pop3Sink_t.error`, `pop3_retrieve`

[Top] [POP3 Functions] [POP3 Functions by Task]

---

pop3_uidList

Returns the message numbers and the corresponding unique ids of the message in the maildrop.

Syntax

```
#include <pop3.h>
int pop3_uidList( pop3Client_t * in_pPOP3 );
```

Parameters The function has the following parameters:

`in_pPOP3`	Pointer to the POP3 client.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function uses the UIDL POP3 protocol command to specify that the LIST command uses unique message identifiers. For more information, see Listing Messages.

This function is mapped to one or more callbacks in the `pop3Sink_t` structure. For more information, see POP3 Function Callback Mapping.

See Also `pop3Sink_t.uidListStart`, `pop3Sink_t.uidList`, `pop3Sink_t.uidListComplete`, `pop3Sink_t.error`, `pop3_uidListA`

[Top] [POP3 Functions] [POP3 Functions by Task]

---

pop3_uidListA

Returns a specified message and its corresponding unique id.

Syntax

```
#include <pop3.h>
int pop3_uidListA( pop3Client_t * in_pPOP3,
                   int in_messageNumber );
```

Parameters The function has the following parameters:

`in_pPOP3`	Pointer to the POP3 client.
`in_messageNumber`	Message number.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function lists all the messages in a mailbox along with their unique identifiers and message numbers. It uses the UIDL POP3 protocol command to specify that the LIST command uses unique message identifiers. For more information, see Listing Messages.

This function is mapped to one or more callbacks in the `pop3Sink_t` structure. For more information, see POP3 Function Callback Mapping.

See Also `pop3Sink_t.uidListStart`, `pop3Sink_t.uidList`, `pop3Sink_t.uidListComplete`, `pop3Sink_t.error`, `pop3_uidList`

[Top] [POP3 Functions] [POP3 Functions by Task]

---

pop3_user

Enters a user name.

Syntax

```
#include <pop3.h>
int pop3_user( pop3Client_t * in_pPOP3,
            const char * in_user );
```

Parameters The function has the following parameters:

`in_pPOP3`	Pointer to the POP3 client.
`in_user`	User name.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function identifies the user or mail drop by name to the server. It sends the USER POP3 protocol command. For more information, see Logging In.

This function is mapped to one or more callbacks in the `pop3Sink_t` structure. For more information, see POP3 Function Callback Mapping.

See Also `pop3Sink_t.user`, `pop3Sink_t.error`, `pop3_pass`

[Top] [POP3 Functions] [POP3 Functions by Task]

pop3_xAuthList

Returns a list of authenticated users.

Syntax

```
#include <pop3.h>
int pop3_xAuthList( pop3Client_t * in_pPOP3 );
```

Parameters The function has the following parameters:

`in_pPOP3` Pointer to the POP3 client.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function sends the XAUTHLIST POP3 protocol command. For more information, see Listing Messages.

This function is mapped to one or more callbacks in the `pop3Sink_t` structure. For more information, see POP3 Function Callback Mapping.

See Also `pop3Sink_t.xAuthListStart`, `pop3Sink_t.xAuthList`, `pop3Sink_t.xAuthListComplete`, `pop3Sink_t.error`, `pop3_xAuthListA`

[Top] [POP3 Functions] [POP3 Functions by Task]

pop3_xAuthListA

Returns an authenticated user for a given message.

Syntax

```
#include <pop3.h>
int pop3_xAuthListA( pop3Client_t * in_pPOP3,
                     int in_messageNumber );
```

Parameters The function has the following parameters:

`in_pPOP3`	Pointer to the POP3 client.
`in_messageNumber`	Message number.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function sends the XAUTHLIST [arg] POP3 protocol command. For more information, see Listing Messages.

This function is mapped to one or more callbacks in the `pop3Sink_t` structure. For more information, see POP3 Function Callback Mapping.

See Also `pop3Sink_t.xAuthListStart`, `pop3Sink_t.xAuthList`, `pop3Sink_t.xAuthListComplete`, `pop3Sink_t.error`, `pop3_xSender`

[Top] [POP3 Functions] [POP3 Functions by Task]

---

pop3_xSender

Gets the email address of the sender of the specified message.

Syntax

```
#include <pop3.h>
int pop3_xSender( pop3Client_t * in_pPOP3,
              int in_messageNumber );
```

Parameters The function has the following parameters:

`in_pPOP3`	Pointer to the POP3 client.
`in_messageNumber`	Message number.

Returns

- If successful, the function returns `NSMAIL_OK`.
- If unsuccessful, the function returns an error message. For a complete list, see "Error Definitions."

Description This function allows a client to query whether an particular message has been authenticated. The server returns the email address of the authenticated sender in a callback or, if no authenticated sender is found, an empty OK string.

This function sends the XSENDER [arg] POP3 protocol command. It is mapped to one or more callbacks in the `pop3Sink_t` structure. For more information, see POP3 Function Callback Mapping.

See Also `pop3Sink_t.xSender`, `pop3Sink_t.error`, `pop3_xAuthList`, `pop3_xAuthListA`

[Top] [POP3 Functions] [POP3 Functions by Task]

POP3 Structures

This section defines POP3 data structures, listed in alphabetical order.

```
pop3Client_t      pop3Sink_t
```

[Top] [POP3 Functions]

---

pop3Client_t

Structure used by a client to communicate with a POP3 server.

Syntax

```
typedef struct pop3Client pop3Client_t;
```

Description The client uses this structure to communicate with an SMTP server. The developer uses this structure to perform SMTP API operations. All data contained within the client structure is used internally by the API.

For more information, see Creating a Client.

See Also `pop3_free`, `pop3_initialize`

[Top] [POP3 Structures] [POP3 Functions]

pop3Sink_t

Definition of the POP3 response sink.

Syntax

```
typedef struct pop3Sink
{
   void * pOpaqueData;
   void (*connect)( pop3SinkPtr_t * in_pPOP3Sink,
      const char * in_responseMessage );
   void (*dele)( pop3SinkPtr_t * in_pPOP3Sink,
      const char * in_responseMessage );
   void (*error)( pop3SinkPtr_t * in_pPOP3Sink,
      const char * in_responseMessage );
   void (*listStart) ( pop3SinkPtr_t * in_pPOP3Sink );
   void (*list)( pop3SinkPtr_t * in_pPOP3Sink,
      int in_messageNumber, int in_octetCount );
   void (*listComplete)(
      pop3SinkPtr_t * in_pPOP3Sink );
   void (*noop)( pop3SinkPtr_t * in_pPOP3Sink );
   void (*pass)(pop3SinkPtr_t * in_pPOP3Sink,
       const char * in_responseMessage );
   void (*quit)(pop3SinkPtr_t * in_pPOP3Sink,
      const char * in_responseMessage );
   void (*reset)(pop3SinkPtr_t * in_pPOP3Sink,
      const char * in_responseMessage );
   void (*retrieveStart)(
      pop3SinkPtr_t * in_pPOP3Sink,
      int in_messageNumber, int in_octetCount );
   void (*retrieve)( pop3SinkPtr_t * in_pPOP3Sink,
      const char * in_messageChunk );
   void (*retrieveComplete) ( pop3SinkPtr_t *
      in_pPOP3Sink );
   void (*sendCommandStart) ( pop3SinkPtr_t *
      in_pPOP3Sink );
   void (*sendCommand) ( pop3SinkPtr_t * in_pPOP3Sink,
      const char * in_responseMessage );
   void (*sendCommandComplete)(
      pop3SinkPtr_t * in_pPOP3Sink );
```

```
   void (*stat)( pop3SinkPtr_t * in_pPOP3Sink,
      int in_messageCount, int in_octetCount );
   void (*topStart)( pop3SinkPtr_t * in_pPOP3Sink,
      int in_messageNumber );
   void (*top)( pop3SinkPtr_t * in_pPOP3Sink,
      const char * in_responseLine );
   void (*topComplete)( pop3SinkPtr_t * in_pPOP3Sink );
   void (*uidListStart)( pop3SinkPtr_t * in_pPOP3Sink );
   void (*uidList)( pop3SinkPtr_t * in_pPOP3Sink,
      int in_messageNumber, const char * in_uid );
   void (*uidListComplete)( pop3SinkPtr_t * in_pPOP3Sink );
   void (*user)( pop3SinkPtr_t * in_pPOP3Sink,
      const char * in_responseMessage );
   void (*xAuthListStart)(
      pop3SinkPtr_t * in_pPOP3Sink );
   void (*xAuthList)( pop3SinkPtr_t * in_pPOP3Sink,
      int in_messageNumber,
      const char * in_emailAddress );
   void (*xAuthListComplete)(
      pop3SinkPtr_t * in_pPOP3Sink );
   void (*xSender)( pop3SinkPtr_t * in_pPOP3Sink,
      const char * in_emailAddress );
} pop3SinkPtr_t;
```

Fields The structure has the following fields:

`void * pOpaqueData;`	User data.
`void (*connect)( pop3SinkPtr_t in_pPOP3Sink, const char * in_responseMessage );`	Notification for the response to the connection to the server. Parameters: the response sink to use, the server's response message.
`void (*dele)( pop3SinkPtr_t in_pPOP3Sink, const char * in_responseMessage );`	Notification for the response to the DELE command. Parameters: the response sink to use, the server's response message.
`void (*error)( pop3SinkPtr_t in_pPOP3Sink, const char * in_responseMessage );`	Notification for an error. Parameters: the response sink to use, the server's error response message.
`void (*listStart)( pop3SinkPtr_t in_pPOP3Sink );`	Notification for the start of the LIST command. Parameters: the response sink to use.
`void (*list)( pop3SinkPtr_t in_pPOP3Sink, int in_messageNumber, int in_octetCount );`	Notification for the response to the LIST command. Parameters: the response sink to use, number of the message, octet count of the message.
`void (*listComplete)( pop3SinkPtr_t in_pPOP3Sink );`	Notification for the completion of the LIST command. Parameters: the response sink to use.
`void (*noop)( pop3SinkPtr_t in_pPOP3Sink );`	Notification for the response to the NOOP command. Parameters: the response sink to use.
`void (*pass)( pop3SinkPtr_t in_pPOP3Sink, const char * in_responseMessage );`	Notification for the response to the PASS command. Parameters: the response sink to use, server response.
`void (*quit)( pop3SinkPtr_t in_pPOP3Sink, const char * in_responseMessage );`	Notification for the response to the QUIT command. Parameters: the response sink to use, server response.
`void (*reset)( pop3SinkPtr_t in_pPOP3Sink, const char * in_responseMessage );`	Notification for the response to the RSET command. Parameters: the response sink to use, server response.
`void (*retrieveStart)( pop3SinkPtr_t in_pPOP3Sink, int in_messageNumber, int in_octetCount );`	Notification for the start of the RETR command. Parameters: the response sink to use, message numbers, octet count of message.

```
void (*retrieve)(
   pop3SinkPtr_t in_pPOP3Sink,
   const char * in_messageChunk );
```

Notification for the response to the RETR command. Parameters: the response sink to use, chunk of message to retrieve

```
void (*retrieveComplete)
   pop3SinkPtr_t in_pPOP3Sink );
```

Notification for the completion of the RETR command. Parameters: the response sink to use.

```
void (*sendCommandStart)
   pop3SinkPtr_t in_pPOP3Sink );
```

Notification for the start of an extended command. Parameters: the response sink to use.

```
void (*sendCommand)(
   pop3SinkPtr_t in_pPOP3Sink,
   const char * in_responseMessage );
```

Notification for the response of an extended command. Parameters: the response sink to use, server response.

```
void (*sendCommandComplete)
   pop3SinkPtr_t in_pPOP3Sink );
```

Notification for the completion of and extended command. Parameters: the response sink to use.

```
void (*stat)(
   pop3SinkPtr_t in_pPOP3Sink,
   int in_messageCount,
   int in_octetCount );
```

Notification for the response to the STAT command. Parameters: the response sink to use, message numbers, octet count of message.

```
void (*topStart)(
   pop3SinkPtr_t * in_pPOP3Sink,
   int in_messageNumber );
```

Notification for the start of the TOP command. Parameters: the response sink to use, number of the message.

```
void (*top)(
   pop3SinkPtr_t in_pPOP3Sink,
   const char * in_responseLine );
```

Notification for the response to the TOP command. Parameters: the response sink to use, line of message.

```
void (*topComplete)(
   pop3SinkPtr_t in_pPOP3Sink );
```

Notification for the completion of the TOP command. Parameters: the response sink to use.

```
void (*uidListStart)(
   pop3SinkPtr_t in_pPOP3Sink );
```

Notification for the start of the UIDL command. Parameters: the response sink to use.

```
void (*uidList)(
   pop3SinkPtr_t in_pPOP3Sink,
   int in_messageNumber,
   const char * in_uid );
```

Notification for the response to the UIDL command. Parameters: the response sink to use, number of the message, unique identifier.

```
void (*uidListComplete)(
   pop3SinkPtr_t in_pPOP3Sink );
```

Notification for the completion of the UIDL command. Parameters: the response sink to use.

`void (*user)( pop3SinkPtr_t in_pPOP3Sink, const char * in_responseMessage );`	Notification for the response to the USER command. Parameters: the response sink to use, server response.
`void (*xAuthListStart)( pop3SinkPtr_t in_pPOP3Sink );`	Notification for the start of the XAUTHLIST command. Parameters: the response sink to use.
`void (*xAuthList)( pop3SinkPtr_t in_pPOP3Sink, int in_messageNumber, const char * in_emailAddress );`	Notification for the response to the XAUTHLIST command. Parameters: the response sink to use, number of the message, email address.
`void (*xAuthListComplete) pop3SinkPtr_t in_pPOP3Sink );`	Notification for the completion of the XAUTHLIST command. Parameters: the response sink to use.
`void (*xSender)( pop3SinkPtr_t in_pPOP3Sink, const char * in_emailAddress );`	Notification for the response to the XSENDER command. Parameters: the response sink to use, the email address.

Description The POP3 response sink structure is made up of function pointers and opaque data. You must define the functions yourself, and you must point these pointers to your functions if you want to receive the related server responses. For more information, see SDK Response Sinks for C and Creating a Response Sink.

The function pointers serve as callbacks for many POP3 functions. See POP3 Function Callback Mapping.

See Also `pop3_initialize, pop3Sink_free`

[Top] [POP3 Structures]

Appendix

A

Writing Multithreaded Applications with the Messaging Access SDK

This appendix provides some important information for developers who want to take advantage of multithreading in their messaging applications.

The C Messaging Access SDK is NOT thread-safe. A multithreaded application must ensure the atomic execution of each function. In addition, it may need to provide a lock on the SDK in order to synchronize state-sensitive sequences of commands within the SMTP, POP3, and IMAP4 modules.

For example, if two threads that are connected to the same SMTP server are used to send mail, a lock is needed to prevent both threads from sending mail at the same time. If the `smtp_rcptTo` calls of the two threads are interleaved, the email may be sent to the wrong destination.

[Top]

Multithreading in the MIME API

In general, in the MIME API, concurrent access by multiple threads is not applicable. This is because multiple threads should not need to change the same `mime_message_t` or other MIME objects, for example, `mime_multiPart_t`, concurrently.

One situation that is applicable in a multithreaded environment is the simultaneous parsing of multiple messages by different threads. To do this, use the `parseEntireMessage` or the `parseEntireMessageInputstream` calls, which can be invoked by multiple threads at the same time.

During MIME dynamic parser operations, multiple threads can share the same instance of the `mimeParser` structure to parse different messages, but the client application must serialize access to the `mimeParser` instance. Several `mimeParser` instances can share a single instance of the `mimeDataSink_t` structure.

[Top]

Index

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

X

Messaging Access SDK Guide