NAME

MIME::Lite − low−calorie MIME generator

WAIT!

MIME::Lite is not recommended by its current maintainer. There are a number of alternatives, like Email::MIME or MIME::Entity and Email::Sender, which you should probably use instead. MIME::Lite continues to accrue weird bug reports, and it is not receiving a large amount of refactoring due to the availability of better alternatives. Please consider using something else.

SYNOPSIS

Create and send using the default send method for your OS a single-part message:

```
use MIME::Lite;
### Create a new single−part message, to send a GIF file:
$msg = MIME::Lite−>new(
    From     => 'me AT myhost DOT com',
    To       => 'you AT yourhost DOT com',
    Cc       => 'some AT other DOT com, some AT more DOT com',
    Subject  => 'Helloooooo, nurse!',
    Type     => 'image/gif',
    Encoding => 'base64',
    Path     => 'hellonurse.gif'
);
$msg−>send; # send via default
```

Create a multipart message (i.e., one with attachments) and send it SMTP

```
### Create a new multipart message:
$msg = MIME::Lite−>new(
    From    => 'me AT myhost DOT com',
    To      => 'you AT yourhost DOT com',
    Cc      => 'some AT other DOT com, some AT more DOT com',
    Subject => 'A message with 2 parts...',
    Type    => 'multipart/mixed'
);

### Add parts (each "attach" has same arguments as "new"):
$msg−>attach(
    Type     => 'TEXT',
    Data     => "Here's the GIF file you wanted"
);
$msg−>attach(
    Type     => 'image/gif',
    Path     => 'aaa000123.gif',
    Filename => 'logo.gif',
    Disposition => 'attachment'
);
### use Net:SMTP to do the sending
$msg−>send('smtp','some.host', Debug=>1 );
```

Output a message:

```
### Format as a string:
$str = $msg−>as_string;

### Print to a filehandle (say, a "sendmail" stream):
$msg−>print(\*SENDMAIL);
```

Send a message:

```
    ### Send in the "best" way (the default is to use "sendmail"):
    $msg->send;
    ### Send a specific way:
    $msg->send('type',@args);
```

Specify default send method:

```
    MIME::Lite->send('smtp','some.host',Debug=>0);
```

with authentication

```
    MIME::Lite->send('smtp','some.host', AuthUser=>$user, AuthPass=>$pass);
```

DESCRIPTION

In the never-ending quest for great taste with fewer calories, we proudly present: *MIME::Lite*.

MIME::Lite is intended as a simple, standalone module for generating (not parsing!) MIME messages... specifically, it allows you to output a simple, decent single- or multi-part message with text or binary attachments. It does not require that you have the Mail:: or MIME:: modules installed, but will work with them if they are.

You can specify each message part as either the literal data itself (in a scalar or array), or as a string which can be given to *open()* to get a readable filehandle (e.g., ''<filename'' or ''somecommand|'').

You don't need to worry about encoding your message data: this module will do that for you. It handles the 5 standard MIME encodings.

EXAMPLES

Create a simple message containing just text

```
    $msg = MIME::Lite->new(
        From     =>'me AT myhost DOT com',
        To       =>'you AT yourhost DOT com',
        Cc       =>'some AT other DOT com, some AT more DOT com',
        Subject  =>'Helloooooo, nurse!',
        Data     =>"How's it goin', eh?"
    );
```

Create a simple message containing just an image

```
    $msg = MIME::Lite->new(
        From     =>'me AT myhost DOT com',
        To       =>'you AT yourhost DOT com',
        Cc       =>'some AT other DOT com, some AT more DOT com',
        Subject  =>'Helloooooo, nurse!',
        Type     =>'image/gif',
        Encoding =>'base64',
        Path     =>'hellonurse.gif'
    );
```

Create a multipart message

```
    ### Create the multipart "container":
    $msg = MIME::Lite->new(
        From    =>'me AT myhost DOT com',
        To      =>'you AT yourhost DOT com',
        Cc      =>'some AT other DOT com, some AT more DOT com',
        Subject =>'A message with 2 parts...',
        Type    =>'multipart/mixed'
    );

    ### Add the text message part:
    ### (Note that "attach" has same arguments as "new"):
    $msg->attach(
        Type     =>'TEXT',
        Data     =>"Here's the GIF file you wanted"
    );
```

```
    ### Add the image part:
    $msg->attach(
        Type        =>'image/gif',
        Path        =>'aaa000123.gif',
        Filename    =>'logo.gif',
        Disposition => 'attachment'
    );
```

Attach a GIF to a text message

This will create a multipart message exactly as above, but using the ''attach to singlepart'' hack:

```
    ### Start with a simple text message:
    $msg = MIME::Lite->new(
        From    =>'me AT myhost DOT com',
        To      =>'you AT yourhost DOT com',
        Cc      =>'some AT other DOT com, some AT more DOT com',
        Subject =>'A message with 2 parts...',
        Type    =>'TEXT',
        Data    =>"Here's the GIF file you wanted"
    );

    ### Attach a part... the make the message a multipart automatically:
    $msg->attach(
        Type     =>'image/gif',
        Path     =>'aaa000123.gif',
        Filename =>'logo.gif'
    );
```

Attach a pre-prepared part to a message

```
    ### Create a standalone part:
    $part = MIME::Lite->new(
        Top      => 0,
        Type     =>'text/html',
        Data     =>'<H1>Hello</H1>',
    );
    $part->attr('content-type.charset' => 'UTF-8');
    $part->add('X-Comment' => 'A message for you');

    ### Attach it to any message:
    $msg->attach($part);
```

Print a message to a filehandle

```
    ### Write it to a filehandle:
    $msg->print(\*STDOUT);

    ### Write just the header:
    $msg->print_header(\*STDOUT);

    ### Write just the encoded body:
    $msg->print_body(\*STDOUT);
```

Print a message into a string

```
    ### Get entire message as a string:
    $str = $msg->as_string;

    ### Get just the header:
    $str = $msg->header_as_string;

    ### Get just the encoded body:
    $str = $msg->body_as_string;
```

Send a message

```
### Send in the "best" way (the default is to use "sendmail"):
$msg->send;
```

Send an HTML document... with images included!

```
$msg = MIME::Lite->new(
     To      =>'you AT yourhost DOT com',
     Subject =>'HTML with in-line images!',
     Type    =>'multipart/related'
);
$msg->attach(
    Type => 'text/html',
    Data => qq{
        <body>
            Here's <i>my</i> image:
            <img src="cid:myimage.gif">
        </body>
    },
);
$msg->attach(
    Type => 'image/gif',
    Id   => 'myimage.gif',
    Path => '/path/to/somefile.gif',
);
$msg->send();
```

Change how messages are sent

```
### Do something like this in your 'main':
if ($I_DONT_HAVE_SENDMAIL) {
   MIME::Lite->send('smtp', $host, Timeout=>60,
       AuthUser=>$user, AuthPass=>$pass);
}

### Now this will do the right thing:
$msg->send;         ### will now use Net::SMTP as shown above
```

PUBLIC INTERFACE

Global configuration

To alter the way the entire module behaves, you have the following methods/options:

MIME::Lite->*field_order()*

When used as a classmethod, this changes the default order in which headers are output for *all* messages. However, please consider using the instance method variant instead, so you won't stomp on other message senders in the same application.

MIME::Lite->*quiet()*

This classmethod can be used to suppress/unsuppress all warnings coming from this module.

MIME::Lite->*send()*

When used as a classmethod, this can be used to specify a different default mechanism for sending message. The initial default is:

```
MIME::Lite->send("sendmail", "/usr/lib/sendmail -t -oi -oem");
```

However, you should consider the similar but smarter and taint-safe variant:

```
MIME::Lite->send("sendmail");
```

Or, for non-Unix users:

```
MIME::Lite->send("smtp");
```

`$MIME::Lite::AUTO_CC`

If true, automatically send to the Cc/Bcc addresses for *send_by_smtp()*. Default is **true**.

`$MIME::Lite::AUTO_CONTENT_TYPE`
If true, try to automatically choose the content type from the file name in `new()`/`build()`. In other words, setting this true changes the default `Type` from `"TEXT"` to `"AUTO"`.

Default is **false**, since we must maintain backwards-compatibility with prior behavior. **Please** consider keeping it false, and just using Type 'AUTO' when you *build()* or *attach()*.

`$MIME::Lite::AUTO_ENCODE`
If true, automatically choose the encoding from the content type. Default is **true**.

`$MIME::Lite::AUTO_VERIFY`
If true, check paths to attachments right before printing, raising an exception if any path is unreadable. Default is **true**.

`$MIME::Lite::PARANOID`
If true, we won't attempt to use MIME::Base64, MIME::QuotedPrint, or MIME::Types, even if they're available. Default is **false**. Please consider keeping it false, and trusting these other packages to do the right thing.

Construction

new [PARAMHASH]
Class method, constructor. Create a new message object.

If any arguments are given, they are passed into `build()`; otherwise, just the empty object is created.

attach PART
attach PARAMHASH...
Instance method. Add a new part to this message, and return the new part.

If you supply a single PART argument, it will be regarded as a MIME::Lite object to be attached. Otherwise, this method assumes that you are giving in the pairs of a PARAMHASH which will be sent into `new()` to create the new part.

One of the possibly-quite-useful hacks thrown into this is the ''attach-to-singlepart'' hack: if you attempt to attach a part (let's call it ''part 1'') to a message that doesn't have a content-type of ''multipart'' or ''message'', the following happens:

- A new part (call it ''part 0'') is made.
- The MIME attributes and data (but *not* the other headers) are cut from the ''self'' message, and pasted into ''part 0''.
- The ''self'' is turned into a ''multipart/mixed'' message.
- The new ''part 0'' is added to the ''self'', and *then* ''part 1'' is added.

One of the nice side-effects is that you can create a text message and then add zero or more attachments to it, much in the same way that a user agent like Netscape allows you to do.

build [PARAMHASH]
Class/instance method, initializer. Create (or initialize) a MIME message object. Normally, you'll use the following keys in PARAMHASH:

```
* Data, FH, or Path      (either one of these, or none if multipart)
* Type                   (e.g., "image/jpeg")
* From, To, and Subject  (if this is the "top level" of a message)
```

The PARAMHASH can contain the following keys:

(fieldname)
Any field you want placed in the message header, taken from the standard list of header fields (you don't need to worry about case):

```
Approved      Encrypted     Received      Sender
Bcc           From          References    Subject
Cc            Keywords      Reply-To      To
Comments      Message-ID    Resent-*      X-*
Content-*     MIME-Version  Return-Path
Date                        Organization
```

To give experienced users some veto power, these fields will be set *after* the ones I set... so be careful: *don't set any MIME fields* (like `Content-type`) unless you know what you're doing!

To specify a fieldname that's *not* in the above list, even one that's identical to an option below, just give it with a trailing `":"`, like `"My-field:"`. When in doubt, that *always* signals a mail field (and it sort of looks like one too).

Data

Alternative to "Path" or "FH". The actual message data. This may be a scalar or a ref to an array of strings; if the latter, the message consists of a simple concatenation of all the strings in the array.

Datestamp

Optional. If given true (or omitted), we force the creation of a `Date:` field stamped with the current date/time if this is a top-level message. You may want this if using *send_by_smtp()*. If you don't want this to be done, either provide your own Date or explicitly set this to false.

Disposition

Optional. The content disposition, `"inline"` or `"attachment"`. The default is `"inline"`.

Encoding

Optional. The content transfer encoding that should be used to encode your data:

```
Use encoding:     | If your message contains:
------------------------------------------------------------
7bit              | Only 7-bit text, all lines <1000 characters
8bit              | 8-bit text, all lines <1000 characters
quoted-printable  | 8-bit text or long lines (more reliable than "8bit")
base64            | Largely non-textual data: a GIF, a tar file, etc.
```

The default is taken from the Type; generally it is ''binary'' (no encoding) for text/*, message/*, and multipart/*, and ''base64'' for everything else. A value of `"binary"` is generally *not* suitable for sending anything but ASCII text files with lines under 1000 characters, so consider using one of the other values instead.

In the case of ''7bit''/''8bit'', long lines are automatically chopped to legal length; in the case of ''7bit'', all 8-bit characters are automatically *removed*. This may not be what you want, so pick your encoding well! For more info, see ''A MIME PRIMER''.

FH *Alternative to "Data" or "Path".* Filehandle containing the data, opened for reading. See ''ReadNow'' also.

Filename

Optional. The name of the attachment. You can use this to supply a recommended filename for the end-user who is saving the attachment to disk. You only need this if the filename at the end of the ''Path'' is inadequate, or if you're using ''Data'' instead of ''Path''. You should *not* put path information in here (e.g., no ''/'' or ''\'' or '':'' characters should be used).

Id *Optional.* Same as setting ''content-id''.

Length

Optional. Set the content length explicitly. Normally, this header is automatically computed, but only under certain circumstances (see ''Benign limitations'').

Path

Alternative to "Data" or "FH". Path to a file containing the data... actually, it can be any *open()*able expression. If it looks like a path, the last element will automatically be treated as

the filename. See ‘‘ReadNow’’ also.

ReadNow

Optional, for use with “Path”. If true, will open the path and slurp the contents into core now. This is useful if the Path points to a command and you don’t want to run the command over and over if outputting the message several times. **Fatal exception** raised if the open fails.

Top

Optional. If defined, indicates whether or not this is a ‘‘top-level’’ MIME message. The parts of a multipart message are *not* top-level. Default is true.

Type

Optional. The MIME content type, or one of these special values (case-sensitive):

```
"TEXT"   means "text/plain"
"BINARY" means "application/octet-stream"
"AUTO"   means attempt to guess from the filename, falling back
         to 'application/octet-stream'.  This is good if you have
         MIME::Types on your system and you have no idea what
         file might be used for the attachment.
```

The default is `"TEXT"`, but it will be `"AUTO"` if you set `$AUTO_CONTENT_TYPE` to true (sorry, but you have to enable it explicitly, since we don’t want to break code which depends on the old behavior).

A picture being worth 1000 words (which is of course 2000 bytes, so it’s probably more of an ‘‘icon’’ than a ‘‘picture’’, but I digress...), here are some examples:

```
$msg = MIME::Lite->build(
    From     => 'yelling AT inter DOT com',
    To       => 'stocking AT fish DOT net',
    Subject  => "Hi there!",
    Type     => 'TEXT',
    Encoding => '7bit',
    Data     => "Just a quick note to say hi!"
);

$msg = MIME::Lite->build(
    From     => 'dorothy@emerald-city.oz',
    To       => 'gesundheit AT edu DOT edu DOT edu',
    Subject  => "A gif for U"
    Type     => 'image/gif',
    Path     => "/home/httpd/logo.gif"
);

$msg = MIME::Lite->build(
    From     => 'laughing AT all DOT of DOT us',
    To       => 'scarlett AT fiddle DOT dee DOT de',
    Subject  => "A gzipp'ed tar file",
    Type     => 'x-gzip',
    Path     => "gzip < /usr/inc/somefile.tar |",
    ReadNow  => 1,
    Filename => "somefile.tgz"
);
```

To show you what’s really going on, that last example could also have been written:

```
    $msg = new MIME::Lite;
    $msg->build(
        Type     => 'x-gzip',
        Path     => "gzip < /usr/inc/somefile.tar |",
        ReadNow  => 1,
        Filename => "somefile.tgz"
    );
    $msg->add(From    => "laughing AT all DOT of DOT us");
    $msg->add(To      => "scarlett AT fiddle DOT dee DOT de");
    $msg->add(Subject => "A gzipp'ed tar file");
```

Setting/getting headers and attributes

add TAG,VALUE

Instance method. Add field TAG with the given VALUE to the end of the header. The TAG will be converted to all-lowercase, and the VALUE will be made ‘‘safe’’ (returns will be given a trailing space).

Beware: any MIME fields you ‘‘add’’ will override any MIME attributes I have when it comes time to output those fields. Normally, you will use this method to add *non-MIME* fields:

```
    $msg->add("Subject" => "Hi there!");
```

Giving VALUE as an arrayref will cause all those values to be added. This is only useful for special multiple-valued fields like ‘‘Received’’:

```
    $msg->add("Received" => ["here", "there", "everywhere"]
```

Giving VALUE as the empty string adds an invisible placeholder to the header, which can be used to suppress the output of the ‘‘Content-*’’ fields or the special ‘‘MIME-Version’’ field. When suppressing fields, you should use *replace()* instead of *add()*:

```
    $msg->replace("Content-disposition" => "");
```

Note: add() is probably going to be more efficient than `replace()`, so you’re better off using it for most applications if you are certain that you don’t need to *delete()* the field first.

Note: the name comes from Mail::Header.

attr ATTR,[VALUE]

Instance method. Set MIME attribute ATTR to the string VALUE. ATTR is converted to all-lowercase. This method is normally used to set/get MIME attributes:

```
    $msg->attr("content-type"         => "text/html");
    $msg->attr("content-type.charset" => "US-ASCII");
    $msg->attr("content-type.name"    => "homepage.html");
```

This would cause the final output to look something like this:

```
    Content-type: text/html; charset=US-ASCII; name="homepage.html"
```

Note that the special empty sub-field tag indicates the anonymous first sub-field.

Giving VALUE as undefined will cause the contents of the named subfield to be deleted.

Supplying no VALUE argument just returns the attribute’s value:

```
    $type = $msg->attr("content-type");        ### returns "text/html"
    $name = $msg->attr("content-type.name");   ### returns "homepage.html"
```

delete TAG

Instance method. Delete field TAG with the given VALUE to the end of the header. The TAG will be converted to all-lowercase.

```
    $msg->delete("Subject");
```

Note: the name comes from Mail::Header.

field_order FIELD,...FIELD

Class/instance method. Change the order in which header fields are output for this object:

```
$msg->field_order('from', 'to', 'content-type', 'subject');
```

When used as a class method, changes the default settings for all objects:

```
MIME::Lite->field_order('from', 'to', 'content-type', 'subject');
```

Case does not matter: all field names will be coerced to lowercase. In either case, supply the empty array to restore the default ordering.

fields

Instance method. Return the full header for the object, as a ref to an array of `[TAG, VALUE]` pairs, where each TAG is all-lowercase. Note that any fields the user has explicitly set will override the corresponding MIME fields that we would otherwise generate. So, don't say...

```
$msg->set("Content-type" => "text/html; charset=US-ASCII");
```

unless you want the above value to override the ''Content-type'' MIME field that we would normally generate.

Note: I called this ''fields'' because the *header()* method of Mail::Header returns something different, but similar enough to be confusing.

You can change the order of the fields: see ''field_order''. You really shouldn't need to do this, but some people have to deal with broken mailers.

filename [FILENAME]

Instance method. Set the filename which this data will be reported as. This actually sets both ''standard'' attributes.

With no argument, returns the filename as dictated by the content-disposition.

get TAG,[INDEX]

Instance method. Get the contents of field TAG, which might have been set with *set()* or *replace()*. Returns the text of the field.

```
$ml->get('Subject', 0);
```

If the optional 0-based INDEX is given, then we return the INDEX'th occurrence of field TAG. Otherwise, we look at the context: In a scalar context, only the first (0th) occurrence of the field is returned; in an array context, *all* occurrences are returned.

Warning: this should only be used with non-MIME fields. Behavior with MIME fields is TBD, and will raise an exception for now.

get_length

Instance method. Recompute the content length for the message *if the process is trivial*, setting the ''content-length'' attribute as a side-effect:

```
$msg->get_length;
```

Returns the length, or undefined if not set.

Note: the content length can be difficult to compute, since it involves assembling the entire encoded body and taking the length of it (which, in the case of multipart messages, means freezing all the sub-parts, etc.).

This method only sets the content length to a defined value if the message is a singlepart with `"binary"` encoding, *and* the body is available either in-core or as a simple file. Otherwise, the content length is set to the undefined value.

Since content-length is not a standard MIME field anyway (that's right, kids: it's not in the MIME RFCs, it's an HTTP thing), this seems pretty fair.

parts

Instance method. Return the parts of this entity, and this entity only. Returns empty array if this entity has no parts.

This is **not** recursive! Parts can have sub-parts; use *parts_DFS()* to get everything.

parts_DFS

Instance method. Return the list of all MIME::Lite objects included in the entity, starting with the entity itself, in depth-first-search order. If this object has no parts, it alone will be returned.

preamble [TEXT]

Instance method. Get/set the preamble string, assuming that this object has subparts. Set it to undef for the default string.

replace TAG,VALUE

Instance method. Delete all occurrences of fields named TAG, and add a new field with the given VALUE. TAG is converted to all-lowercase.

Beware the special MIME fields (MIME-version, Content−*): if you ''replace'' a MIME field, the replacement text will override the *actual* MIME attributes when it comes time to output that field. So normally you use *attr()* to change MIME fields and *add()*/*replace()* to change *non-MIME* fields:

```
$msg->replace("Subject" => "Hi there!");
```

Giving VALUE as the *empty string* will effectively *prevent* that field from being output. This is the correct way to suppress the special MIME fields:

```
$msg->replace("Content-disposition" => "");
```

Giving VALUE as *undefined* will just cause all explicit values for TAG to be deleted, without having any new values added.

Note: the name of this method comes from Mail::Header.

scrub

Instance method. **This is Alpha code. If you use it, please let me know how it goes.** Recursively goes through the ''parts'' tree of this message and tries to find MIME attributes that can be removed. With an array argument, removes exactly those attributes; e.g.:

```
$msg->scrub(['content-disposition', 'content-length']);
```

Is the same as recursively doing:

```
$msg->replace('Content-disposition' => '');
$msg->replace('Content-length'      => '');
```

Setting/getting message data

binmode [OVERRIDE]

Instance method. With no argument, returns whether or not it thinks that the data (as given by the ''Path'' argument of `build()`) should be read using *binmode()* (for example, when `read_now()` is invoked).

The default behavior is that any content type other than `text/*` or `message/*` is binmode'd; this should in general work fine.

With a defined argument, this method sets an explicit ''override'' value. An undefined argument unsets the override. The new current value is returned.

data [DATA]

Instance method. Get/set the literal DATA of the message. The DATA may be either a scalar, or a reference to an array of scalars (which will simply be joined).

Warning: setting the data causes the ''content-length'' attribute to be recomputed (possibly to nothing).

fh [FILEHANDLE]

Instance method. Get/set the FILEHANDLE which contains the message data.

Takes a filehandle as an input and stores it in the object. This routine is similar to *path()*; one important difference is that no attempt is made to set the content length.

path [PATH]

Instance method. Get/set the PATH to the message data.

Warning: setting the path recomputes any existing ''content-length'' field, and re-sets the ''filename'' (to the last element of the path if it looks like a simple path, and to nothing if not).

resetfh [FILEHANDLE]
Instance method. Set the current position of the filehandle back to the beginning. Only applies if you used ''FH'' in *build()* or *attach()* for this message.

Returns false if unable to reset the filehandle (since not all filehandles are seekable).

read_now
Instance method. Forces data from the path/filehandle (as specified by `build()`) to be read into core immediately, just as though you had given it literally with the `Data` keyword.

Note that the in-core data will always be used if available.

Be aware that everything is slurped into a giant scalar: you may not want to use this if sending tar files! The benefit of *not* reading in the data is that very large files can be handled by this module if left on disk until the message is output via `print()` or `print_body()`.

sign PARAMHASH
Instance method. Sign the message. This forces the message to be read into core, after which the signature is appended to it.

Data
As in `build()`: the literal signature data. Can be either a scalar or a ref to an array of scalars.

Path
As in `build()`: the path to the file.

If no arguments are given, the default is:

```
Path => "$ENV{HOME}/.signature"
```

The content-length is recomputed.

verify_data
Instance method. Verify that all ''paths'' to attached data exist, recursively. It might be a good idea for you to do this before a *print()*, to prevent accidental partial output if a file might be missing. Raises exception if any path is not readable.

Output

print [OUTHANDLE]
Instance method. Print the message to the given output handle, or to the currently-selected filehandle if none was given.

All OUTHANDLE has to be is a filehandle (possibly a glob ref), or any object that responds to a *print()* message.

print_body [OUTHANDLE] [IS_SMTP]
Instance method. Print the body of a message to the given output handle, or to the currently-selected filehandle if none was given.

All OUTHANDLE has to be is a filehandle (possibly a glob ref), or any object that responds to a *print()* message.

Fatal exception raised if unable to open any of the input files, or if a part contains no data, or if an unsupported encoding is encountered.

IS_SMPT is a special option to handle SMTP mails a little more intelligently than other send mechanisms may require. Specifically this ensures that the last byte sent is NOT '\n' (octal \012) if the last two bytes are not '\r\n' (\015\012) as this will cause some SMTP servers to hang.

print_header [OUTHANDLE]
Instance method. Print the header of the message to the given output handle, or to the currently-selected filehandle if none was given.

All OUTHANDLE has to be is a filehandle (possibly a glob ref), or any object that responds to a *print()* message.

as_string
Instance method. Return the entire message as a string, with a header and an encoded body.

body_as_string
Instance method. Return the encoded body as a string. This is the portion after the header and the blank line.

Note: actually prepares the body by "printing" to a scalar. Proof that you can hand the `print*()` methods any blessed object that responds to a `print()` message.

header_as_string
Instance method. Return the header as a string.

Sending

send
send HOW, HOWARGS...
Class/instance method. This is the principal method for sending mail, and for configuring how mail will be sent.

As a class method with a HOW argument and optional HOWARGS, it sets the default sending mechanism that the no-argument instance method will use. The HOW is a facility name (**see below**), and the HOWARGS is interpreted by the facility. The class method returns the previous HOW and HOWARGS as an array.

```
MIME::Lite->send('sendmail', "d:\\programs\\sendmail.exe");
...
$msg = MIME::Lite->new(...);
$msg->send;
```

As an instance method with arguments (a HOW argument and optional HOWARGS), sends the message in the requested manner; e.g.:

```
$msg->send('sendmail', "d:\\programs\\sendmail.exe");
```

As an instance method with no arguments, sends the message by the default mechanism set up by the class method. Returns whatever the mail-handling routine returns: this should be true on success, false/exception on error:

```
$msg = MIME::Lite->new(From=>...);
$msg->send || die "you DON'T have mail!";
```

On Unix systems (or rather non-Win32 systems), the default setting is equivalent to:

```
MIME::Lite->send("sendmail", "/usr/lib/sendmail -t -oi -oem");
```

On Win32 systems the default setting is equivalent to:

```
MIME::Lite->send("smtp");
```

The assumption is that on Win32 your site/lib/Net/libnet.cfg file will be preconfigured to use the appropriate SMTP server. See below for configuring for authentication.

There are three facilities:

"sendmail", ARGS...
Send a message by piping it into the "sendmail" command. Uses the *send_by_sendmail()* method, giving it the ARGS. This usage implements (and deprecates) the `sendmail()` method.

"smtp", [HOSTNAME, [NAMEDPARMS]]
Send a message by SMTP, using optional HOSTNAME as SMTP-sending host. Net::SMTP will be required. Uses the *send_by_smtp()* method. Any additional arguments passed in will also be passed through to send_by_smtp. This is useful for things like mail servers requiring authentication where you can say something like the following

```
MIME::Lite->send('smtp', $host, AuthUser=>$user, AuthPass=>$pass);
```

which will configure things so future uses of

```
$msg->send();
```

do the right thing.

''sub'', \&SUBREF, ARGS...
Sends a message MSG by invoking the subroutine SUBREF of your choosing, with MSG as the first argument, and ARGS following.

For example: let's say you're on an OS which lacks the usual Unix ''sendmail'' facility, but you've installed something a lot like it, and you need to configure your Perl script to use this ''sendmail.exe'' program. Do this following in your script's setup:

```
MIME::Lite->send('sendmail', "d:\\programs\\sendmail.exe");
```

Then, whenever you need to send a message `$msg`, just say:

```
$msg->send;
```

That's it. Now, if you ever move your script to a Unix box, all you need to do is change that line in the setup and you're done. All of your `$msg->send` invocations will work as expected.

After sending, the method *last_send_successful()* can be used to determine if the send was successful or not.

send_by_sendmail SENDMAILCMD
send_by_sendmail PARAM=>VALUE, ARRAY, HASH...
Instance method. Send message via an external ''sendmail'' program (this will probably only work out-of-the-box on Unix systems).

Returns true on success, false or exception on error.

You can specify the program and all its arguments by giving a single string, SENDMAILCMD. Nothing fancy is done; the message is simply piped in.

However, if your needs are a little more advanced, you can specify zero or more of the following PARAM/VALUE pairs (or a reference to hash or array of such arguments as well as any combination thereof); a Unix-style, taint-safe ''sendmail'' command will be constructed for you:

Sendmail
Full path to the program to use. Default is ''/usr/lib/sendmail''.

BaseArgs
Ref to the basic array of arguments we start with. Default is `["-t", "-oi", "-oem"]`.

SetSender
Unless this is *explicitly* given as false, we attempt to automatically set the `-f` argument to the first address that can be extracted from the ''From:'' field of the message (if there is one).

What is the -f, and why do we use it? Suppose we did *not* use `-f`, and you gave an explicit ''From:'' field in your message: in this case, the sendmail ''envelope'' would indicate the *real* user your process was running under, as a way of preventing mail forgery. Using the `-f` switch causes the sender to be set in the envelope as well.

So when would I NOT want to use it? If sendmail doesn't regard you as a ''trusted'' user, it will permit the `-f` but also add an ''X-Authentication-Warning'' header to the message to indicate a forged envelope. To avoid this, you can either (1) have SetSender be false, or (2) make yourself a trusted user by adding a `T` configuration
command to your *sendmail.cf* file
(e.g.: `Teryq` if the script is running as user ''eryq'').

FromSender
If defined, this is identical to setting SetSender to true, except that instead of looking at the ''From:'' field we use the address given by this option. Thus:

```
FromSender => 'me AT myhost DOT com'
```

After sending, the method *last_send_successful()* can be used to determine if the send was successful or not.

send_by_smtp HOST, ARGS...
send_by_smtp REF, HOST, ARGS
Instance method. Send message via SMTP, using Net::SMTP — which will be required for this feature.

HOST is the name of SMTP server to connect to, or undef to have Net::SMTP use the defaults in Libnet.cfg.

ARGS are a list of key value pairs which may be selected from the list below. Many of these are just passed through to specific Net::SMTP commands and you should review that module for details.

Please see Good-vs-bad email addresses with *send_by_smtp()*

Hello
LocalAddr
LocalPort
Timeout
Port
ExactAddresses
Debug
See *Net::SMTP::new()* for details.

Size
Return
Bits
Transaction
Envelope
See *Net::SMTP::mail()* for details.

SkipBad
If true doesn't throw an error when multiple email addresses are provided and some are not valid. See *Net::SMTP::recipient()* for details.

AuthUser
Authenticate with *Net::SMTP::auth()* using this username.

AuthPass
Authenticate with *Net::SMTP::auth()* using this password.

NoAuth
Normally if AuthUser and AuthPass are defined MIME::Lite will attempt to use them with the *Net::SMTP::auth()* command to authenticate the connection, however if this value is true then no authentication occurs.

To Sets the addresses to send to. Can be a string or a reference to an array of strings. Normally this is extracted from the To: (and Cc: and Bcc: fields if `$AUTO_CC` is true).

This value overrides that.

From
Sets the email address to send from. Normally this value is extracted from the Return-Path: or From: field of the mail itself (in that order).

This value overrides that.

Returns: True on success, croaks with an error message on failure.

After sending, the method *last_send_successful()* can be used to determine if the send was successful or not.

send_by_testfile FILENAME
Instance method. Print message to a file (namely FILENAME), which will default to mailer.testfile If file exists, message will be appended.

last_send_successful
This method will return TRUE if the last *send()* or *send_by_XXX()* method call was successful. It will return defined but false if it was not successful, and undefined if the object had not been used to send yet.

sendmail COMMAND...
Class method, DEPRECATED. Declare the sender to be ''sendmail'', and set up the ''sendmail'' command. *You should use send() instead.*

Miscellaneous

quiet ONOFF

Class method. Suppress/unsuppress all warnings coming from this module.

```
MIME::Lite->quiet(1);       ### I know what I'm doing
```

I recommend that you include that comment as well. And while you type it, say it out loud: if it doesn't feel right, then maybe you should reconsider the whole line. `;-)`

NOTES

How do I prevent "Content" headers from showing up in my mail reader?

Apparently, some people are using mail readers which display the MIME headers like ''Content-disposition'', and they want MIME::Lite not to generate them ''because they look ugly''.

Sigh.

Y'know, kids, those headers aren't just there for cosmetic purposes. They help ensure that the message is *understood* correctly by mail readers. But okay, you asked for it, you got it... here's how you can suppress the standard MIME headers. Before you send the message, do this:

```
$msg->scrub;
```

You can *scrub()* any part of a multipart message independently; just be aware that it works recursively. Before you scrub, note the rules that I follow:

Content-type

You can safely scrub the ''content-type'' attribute if, and only if, the part is of type ''text/plain'' with charset ''us-ascii''.

Content-transfer-encoding

You can safely scrub the ''content-transfer-encoding'' attribute if, and only if, the part uses ''7bit'', ''8bit'', or ''binary'' encoding. You are far better off doing this if your lines are under 1000 characters. Generally, that means you *can* scrub it for plain text, and you can *not* scrub this for images, etc.

Content-disposition

You can safely scrub the ''content-disposition'' attribute if you trust the mail reader to do the right thing when it decides whether to show an attachment inline or as a link. Be aware that scrubbing both the content-disposition and the content-type means that there is no way to ''recommend'' a filename for the attachment!

Note: there are reports of brain-dead MUAs out there that do the wrong thing if you *provide* the content-disposition. If your attachments keep showing up inline or vice-versa, try scrubbing this attribute.

Content-length

You can always scrub ''content-length'' safely.

How do I give my attachment a [different] recommended filename?

By using the Filename option (which is different from Path!):

```
$msg->attach(Type => "image/gif",
             Path => "/here/is/the/real/file.GIF",
             Filename => "logo.gif");
```

You should *not* put path information in the Filename.

Benign limitations

This is ''lite'', after all...

- There's no parsing. Get MIME-tools if you need to parse MIME messages.
- MIME::Lite messages are currently *not* interchangeable with either Mail::Internet or MIME::Entity objects. This is a completely separate module.
- A content-length field is only inserted if the encoding is binary, the message is a singlepart, and all the document data is available at `build()` time by virtue of residing in a simple path, or in-core. Since content-length is not a standard MIME field anyway (that's right, kids: it's not in the MIME RFCs, it's an HTTP thing), this seems pretty fair.

- MIME::Lite alone cannot help you lose weight. You must supplement your use of MIME::Lite with a healthy diet and exercise.

Cheap and easy mailing

I thought putting in a default ‘‘sendmail’’ invocation wasn’t too bad an idea, since a lot of Perlers are on UNIX systems. (As of version 3.02 this is default only on Non−Win32 boxen. On Win32 boxen the default is to use SMTP and the defaults specified in the site/lib/Net/libnet.cfg)

The out-of-the-box configuration is:

```
MIME::Lite->send('sendmail', "/usr/lib/sendmail -t -oi -oem");
```

By the way, these arguments to sendmail are:

```
-t      Scan message for To:, Cc:, Bcc:, etc.

-oi     Do NOT treat a single "." on a line as a message terminator.
        As in, "-oi vey, it truncated my message... why?!"

-oem    On error, mail back the message (I assume to the
        appropriate address, given in the header).
        When mail returns, circle is complete.  Jai Guru Deva -oem.
```

Note that these are the same arguments you get if you configure to use the smarter, taint-safe mailing:

```
MIME::Lite->send('sendmail');
```

If you get ‘‘X−Authentication-Warning’’ headers from this, you can forgo diddling with the envelope by instead specifying:

```
MIME::Lite->send('sendmail', SetSender=>0);
```

And, if you’re not on a Unix system, or if you’d just rather send mail some other way, there’s always SMTP, which these days probably requires authentication so you probably need to say

```
MIME::Lite->send('smtp', "smtp.myisp.net",
   AuthUser=>"YourName",AuthPass=>"YourPass" );
```

Or you can set up your own subroutine to call. In any case, check out the *send()* method.

WARNINGS

Good-vs-bad email addresses with *send_by_smtp()*

If using *send_by_smtp()*, be aware that unless you explicitly provide the email addresses to send to and from you will be forcing MIME::Lite to extract email addresses out of a possible list provided in the `To:`, `Cc:`, and `Bcc:` fields. This is tricky stuff, and as such only the following sorts of addresses will work reliably:

```
username
full DOT name AT some DOT host DOT com
"Name, Full" <full DOT name AT some DOT host DOT com>
```

Disclaimer: MIME::Lite was never intended to be a Mail User Agent, so please don’t expect a full implementation of RFC−822. Restrict yourself to the common forms of Internet addresses described herein, and you should be fine. If this is not feasible, then consider using MIME::Lite to *prepare* your message only, and using Net::SMTP explicitly to *send* your message.

Note: As of MIME::Lite v3.02 the mail name extraction routines have been beefed up considerably. Furthermore if Mail::Address if provided then name extraction is done using that. Accordingly the above advice is now less true than it once was. Funky email names *should* work properly now. However the disclaimer remains. Patches welcome. :−)

Formatting of headers delayed until *print()*

This class treats a MIME header in the most abstract sense, as being a collection of high-level attributes. The actual RFC−822−style header fields are not constructed until it’s time to actually print the darn thing.

Encoding of data delayed until *print()*

When you specify message bodies (in *build()* or *attach()*) — whether by **FH**, **Data**, or **Path** — be warned that we don’t attempt to open files, read filehandles, or encode the data until *print()* is invoked.

In the past, this created some confusion for users of sendmail who gave the wrong path to an attachment body, since enough of the *print()* would succeed to get the initial part of the message out. Nowadays, `$AUTO_VERIFY` is used to spot-check the Paths given before the mail facility is employed. A whisker slower, but tons safer.

Note that if you give a message body via FH, and try to *print()* a message twice, the second *print()* will not do the right thing unless you explicitly rewind the filehandle.

You can get past these difficulties by using the **ReadNow** option, provided that you have enough memory to handle your messages.

MIME attributes are separate from header fields!

Important: the MIME attributes are stored and manipulated separately from the message header fields; when it comes time to print the header out, *any explicitly-given header fields override the ones that would be created from the MIME attributes.* That means that this:

```
### DANGER ### DANGER ### DANGER ### DANGER ### DANGER ###
$msg->add("Content-type", "text/html; charset=US-ASCII");
```

will set the exact `"Content-type"` field in the header I write, *regardless of what the actual MIME attributes are.*

This feature is for experienced users only, as an escape hatch in case the code that normally formats MIME header fields isn't doing what you need. And, like any escape hatch, it's got an alarm on it: MIME::Lite will warn you if you attempt to `set()` or `replace()` any MIME header field. Use `attr()` instead.

Beware of lines consisting of a single dot

Julian Haight noted that MIME::Lite allows you to compose messages with lines in the body consisting of a single ''.''. This is true: it should be completely harmless so long as ''sendmail'' is used with the -oi option (see ''Cheap and easy mailing'').

However, I don't know if using Net::SMTP to transfer such a message is equally safe. Feedback is welcomed.

My perspective: I don't want to magically diddle with a user's message unless absolutely positively necessary. Some users may want to send files with ''.'' alone on a line; my well-meaning tinkering could seriously harm them.

Infinite loops may mean tainted data!

Stefan Sautter noticed a bug in 2.106 where a m//gc match was failing due to tainted data, leading to an infinite loop inside MIME::Lite.

I am attempting to correct for this, but be advised that my fix will silently untaint the data (given the context in which the problem occurs, this should be benign: I've labelled the source code with UNTAINT comments for the curious).

So: don't depend on taint-checking to save you from outputting tainted data in a message.

Don't tweak the global configuration

Global configuration variables are bad, and should go away. Until they do, please follow the hints with each setting on how *not* to change it.

A MIME PRIMER

Content types

The ''Type'' parameter of `build()` is a *content type*. This is the actual type of data you are sending. Generally this is a string of the form `"majortype/minortype"`.

Here are the major MIME types. A more-comprehensive listing may be found in RFC-2046.

application
: Data which does not fit in any of the other categories, particularly data to be processed by some type of application program. `application/octet-stream`, `application/gzip`, `application/postscript`...

audio
: Audio data. `audio/basic`...

image
Graphics data. `image/gif`, `image/jpeg`...

message
A message, usually another mail or MIME message. `message/rfc822`...

multipart
A message containing other messages. `multipart/mixed`, `multipart/alternative`...

text Textual data, meant for humans to read. `text/plain`, `text/html`...

video
Video or video+audio data. `video/mpeg`...

Content transfer encodings

The ‘‘Encoding’’ parameter of `build()`. This is how the message body is packaged up for safe transit.

Here are the 5 major MIME encodings. A more-comprehensive listing may be found in RFC−2045.

7bit
Basically, no *real* encoding is done. However, this label guarantees that no 8−bit characters are present, and that lines do not exceed 1000 characters in length.

8bit
Basically, no *real* encoding is done. The message might contain 8−bit characters, but this encoding guarantees that lines do not exceed 1000 characters in length.

binary
No encoding is done at all. Message might contain 8−bit characters, and lines might be longer than 1000 characters long.

The most liberal, and the least likely to get through mail gateways. Use sparingly, or (better yet) not at all.

base64
Like ‘‘uuencode’’, but very well-defined. This is how you should send essentially binary information (tar files, GIFs, JPEGs, etc.).

quoted-printable
Useful for encoding messages which are textual in nature, yet which contain non-ASCII characters (e.g., Latin−1, Latin−2, or any other 8−bit alphabet).

HELPER MODULES

MIME::Lite works nicely with other certain other modules if they are present. Good to have installed are the latest MIME::Types, Mail::Address, MIME::Base64, MIME::QuotedPrint, and Net::SMTP. Email::Date::Format is strictly required.

If they aren’t present then some functionality won’t work, and other features wont be as efficient or up to date as they could be. Nevertheless they are optional extras.

BUNDLED GOODIES

MIME::Lite comes with a number of extra files in the distribution bundle. This includes examples, and utility modules that you can use to get yourself started with the module.

The ./examples directory contains a number of snippets in prepared form, generally they are documented, but they should be easy to understand.

The ./contrib directory contains a companion/tool modules that come bundled with MIME::Lite, they don’t get installed by default. Please review the POD they come with.

BUGS

The whole reason that version 3.0 was released was to ensure that MIME::Lite is up to date and patched. If you find an issue please report it.

As far as I know MIME::Lite doesn’t currently have any serious bugs, but my usage is hardly comprehensive.

Having said that there are a number of open issues for me, mostly caused by the progress in the community as whole since Eryq last released. The tests are based around an interesting but non standard test framework. I’d like to change it over to using Test::More.

Should tests fail please review the ./testout directory, and in any bug reports please include the output of the relevant file. This is the only redeeming feature of not using Test::More that I can see.

Bug fixes / Patches / Contribution are welcome, however I probably won't apply them unless they also have an associated test. This means that if I don't have the time to write the test the patch wont get applied, so please, include tests for any patches you provide.

VERSION

Version: 3.030

CHANGE LOG

Moved to ./changes.pod

NOTE: Users of the ''advanced features'' of 3.01_0x smtp sending should take care: These features have been REMOVED as they never really fit the purpose of the module. Redundant SMTP delivery is a task that should be handled by another module.

TERMS AND CONDITIONS

NUTRITIONAL INFORMATION

For some reason, the US FDA says that this is now required by law on any products that bear the name ''Lite''...

Version 3.0 is now new and improved! The distribution is now 30% smaller!

```
MIME::Lite                |
----------------------------------------------------------
Serving size:             | 1 module
Servings per container:   | 1
Calories:                 | 0
Fat:                      | 0g
  Saturated Fat:          | 0g
```

Warning: for consumption by hardware only! May produce indigestion in humans if taken internally.

AUTHOR

Eryq (*eryq AT zeegee DOT com*). President, ZeeGee Software Inc. (*http://www.zeegee.com*).

Go to *http://www.cpan.org* for the latest downloads and on-line documentation for this module. Enjoy.

Patches And Maintenance by Yves Orton and many others. Consult ./changes.pod