Sending MIME Email With PHP

Tired of sending those drab textual notifications and newsletters to your friend and clients? Ever wanted to send attachments and/or HTML embedded email.
The answer is MIME. The ensuing few pages explain the basics of MIME, creating MIME-compliant messages and then ends with a working class implementation of sending MIME complaint email in PHP. Note that references to calling script, caller etc. denote the script that uses the class we are about to develop and client/MUA etc. denote a mail reading client or mail user agent.

Some MIME Basics

MIME stand for Multipurpose Internet Mail Extensions. MIME extends the basic text-oriented Internet mail system in order that messages can contain binary attachments.
MIME takes advantage of the fact that RFC 822 places little restriction on the message body content: *the only stipulation is that plain ASCII text be used*. Thus, MIME messages consist of normal Internet text mail with some special RFC 822 compliant headers and formatted bodies (with the attachments represented in a subset of ASCII). These MIME headers give a special meaning to the presence of attachments in your mail.

Anatomy of a MIME Message

An ordinary text e-mail message contains a header portion (To: From: Subject: etc.) and a body portion (Hello Mr., etc.). In a MIME compliant message, there are also headers involved and not surprisingly, parts of the email called MIME parts, prefixed by special headers also. A MIME mail is just an extension of the RFC 822 based email. It however has its own set of RFCs.

Header Fields

MIME headers are broadly classified as MIME Message Headers and MIME Part Headers, based on their placement in an email packet.
The MIME Message Headers are:

- MIME-Version:
  This header provides the version number of MIME used. The value to be used is 1.0.
- Content-Type:
  This identifies the type of data so that it can be handled suitably. Valid types are text, image, audio, video, applications, multipart and message. Note that arbitrary binary attachments must be called `application/octet-stream`. Some examples of this header are image/jpg, application/msword, multipart/mixed, to cite a few.
- Content-Transfer-Encoding:
  This is the most important of all the headers as it shows the type of encoding performed on the data and is used by the client/MUA to decode the attachment. It can take one of 7bit, 8bit, binary, quoted-printable, base64 and custom, per attachment. 7bit encoding is the normal encoding used on US ASCII characters, i.e., retain it as it is. 8bit and binary encoding are not used in general. Quoted printable is used for human-readable normal text if it has to be protected in transit through a gateway that will affect the formatting. Base64 is a common denominator and gives a lazy approach to deciding which encoding to use; this is normally used on binary, non-textual data. Note that any non-7bit data must be encoded using a scheme such it can traverse Internet mail gateways!
- Content-ID:
  This header is useful if Content-Type is message/external-body or multipart/alternative. This is outside the scope of this article.
- Content-Description:
  This is an optional header. Free text descriptions of the contents of any message part. The descriptions must be us-ascii.
- Content-Disposition:
  An experimental header, it is used to provide hints to the client/MUA whether to display attachments inline or as an attachment.

MIME part headers (headers that appear in the actual MIME "attachment" part), can have any of the above except the MIME-Version header. If a MIME header is part of the message block, it applies to the entire message. E.g., if Content-Transfer-Encoding appears in the Message header, it applies to the entire message body, but if it appears under a MIME part, it is applicable to **only** that part.

"Okay, how do I create MIME Compliant Messages?"

With the above general description, let us now look at what exactly is in this so-called MIME Message!

The Simplest MIME Message

This message has no parts to it, i.e., no "attachments". Nevertheless, for it to be a MIME message, it must have the necessary headers.

```
From: php@php.net
To: 'Alex (the Great)' <alex@greece.net>
Subject: Bucephalus
MIME-Version: 1.0

Hello Alexander,

How's Bucephalus doing?
```

This is nothing but a simple RFC-822 compliant message (text email) with the MIME header. Note that if no Content-Type header is specified, `Content-Type: text/plain; charset='us-ascii'` is assumed! Of course, it could be something slightly more complex like the following:

```
From: 'Alex (the Great)' <alex@greece.net>
To: php@php.net
Subject: re: Bucephalus
MIME-Version: 1.0
Content-Type: image/jpg;
        name='buce.jpg'
Content-Transfer-Encoding: base64
Content-Description: Take a look at him yourself

<.....base64 encoded jpg image of Bucephalus...>
```

"Hey, but I want to send a word document and also a picture of my doggie in the same email...!" goes one user! True enough, the sample shown above is too simple and does not have enough pulp to support the fancier and necessary aspects of modern day emailing. In fact, many mail clients will not even show the description field!
This is where we encounter what is called "multipart messages".

Multipart Messages

This concept allows us to send multiple items in the same email. For instance, let us assume that Alexander wanted to email php@php.net a picture of his horse, with its pedigree chart along with a polished note in the email! Such a simple requirement cannot be satisfied without the concept of multipart messages. In such a situation, we create a wrapper using the Content-Type Message header to "hold" the various part of the email, so that the recipient gets the picture, the pedigree chart and the nice note!
The Content-Type header now has a value "multipart" to indicate that it is a complete email message and that the header merely encapsulates the information. Moreover, it has a subtype of "mixed" (after all an image, a pedigree chart and 7bit text message are not the same type, correct?).
Let us look at what the whole picture looks like:

```
From: 'Alex (the Great)' <alex@greece.net>
To: php@php.net
Subject: re: Bucephalus
MIME-Version: 1.0
Content-Type: multipart/mixed;
        boundary="XX-1234DED00099A";
Content-Transfer-Encoding: 7bit

This is a MIME Encoded Message

--XX-1234DED00099A
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit

Hi PHP,

Attached you will find my horse, Bucephalus', pedigree chart and photo.

Alex

--XX-1234DED00099A
Content-Type: image/jpg;
        name="buce.jpg";
Content-Transfer-Encoding: base64
Content-Description: "A photo of Bucephalus"

<.....base64 encoded jpg image of Bucephalus...>

--XX-1234DED00099A
Content-Type: application/octet-stream;
        name="pedigree.doc"
```

```
Content-Transfer-Encoding: base64
Content-Description: "Pedigree Chart of the great horse"

<.....base64 encoded doc (pedigree.doc) of Bucephalus...>

--XX-1234DED00099A-
```

Phew! Looks like a lot of work, does it not? Anyway, let us go over the details:

1. if you notice the Content-Transfer-Encoding on the MIME Message Header, it says "7bit". Since the Content-Type is multipart/mixed, the encoding has to be one of 7bit, 8bit or binary, 7bit being the widely used format.
2. A message like this has a variety of information bundled in it. How will the client "know" to differentiate between the JPG image, the document and the plain text? You will notice a boundary="XX-1234DED00099A"; parameter after the Content-Type. This value tells apart the various contents of the mail. This is called the MIME Boundary Marker. The value for the boundary marker must be as unique as possible to avoid confusions as to the extent (scope) of the mail.
3. The "warning" message is there in order that non-MIME compliant clients can display it to the user, who might otherwise not understand the purport of a blank email
4. Now, back to the boundary marker. If you observe in the sample mail, the boundary marker (XX-1234DED00099A) occurs before ever y part, i.e., there is a boundary marker used between parts, however, they start with two hyphens. An important point to note is that after the last MIME part, the boundary marker not only starts with two hyphens but also ends in two hyphens. This is something that must not be left out for it defines the scope.
5. Let us look at the first two MIME message parts:
    - The first is the plain text message, hence a Content-Type of text/plain and encoding of 7bit (we might have as well omitted as it is understood if not specified).
    - The second is the JPEG image. It says, aptly, Content-Type: image/jpg. The name="buce.jpg" (that appears after Content-Type and called a parameter), specifies the name of the file; this is what one will see as the attachment name in the client. If the name="buce.jpg" were not given, the *might* display the description field (if given) as the name of the attachment (this is not a consistent behavior among all clients, though).
6. Note that the JPEG image can be displayed in the body of the email if the client is capable of displaying inline attachments. Alternatively, you can indicate to the client as to how you want the attachments to appear. For instance if there were a

    ```
    Content-Disposition: attachment
    ```

    header, the JPEG image will be shown as an attachment icon.

The MIME Class

With these basics in mind, let us build and implement a MIME mail class in PHP. In our PHP arsenal, we already have the necessary tools to do the encoding!
The MIME class must have the ability to:

1. Add attachment(s).
2. Perform encoding on the attached data, as per the individual request.
3. Build the MIME parts/headers.
4. Generate a complete email with the MIME parts/headers included.
5. Return the completed mail as string.
6. Send it using the local mailer (or optionally call an SMTP mailer).

The class is called **MIME_mail**. We will discuss the class methods to bridge the gap between the theory and practice. (Reading suggestion: Luis Argerich's OO Programming in PHP). Most of the comments have been stripped off for readability. Some method and class member variables are for internal access only and have been indicated so in the commentary below (and in the original class file).

```php
<?php

class MIME_mail {
 //public:
  var $to;
  var $from;
  var $subject;
  var $body;
  var $headers = "";
  var $errstr="";

  var $base64_func= '';    // if !specified use PHP's base64
  var $qp_func = '';        // None at this time

  var $mailer = "";
    // Set this to the name of a valid mail object
```

```
?>
```

These are public access variables (i.e., you can directly manipulate this in your script). Most of these variables are self-explanatory. `$headers` contains the optional headers you might want to send to the mailer (more on this later). `$errstr` is a variable that contains a readable error string that can be used in the calling script.
`$base64_func` and `$qp_func` are "function handlers" that the users can customize. By default, they are set to null strings. For `$base64_func`, a null string means that we will be using the PHP built-in base64_encode() function...(yeah! Neat isn't it!). Quoted Printable is accessible by the variable`$qp_func`. There is no native quoted-printable encode function in PHP (however, imap enabled installations can use `imap_qprint()`). We will not be discussing the quoted-printable encoding method in this article.

```
<?php

//private:
  var $mimeparts = array();

?>
```

`$mimeparts` is an internal array that contains the individual MIME-compliant parts of the email message. Please desist from manipulating this and other private methods/variables outside this (or other inherited) class.

```
<?php

// Constructor.
 function MIME_mail($from="", $to="", $subject="", $body="", $headers = "") {
    $this->to = $to;
    $this->from = $from;
    $this->subject = $subject;
    $this->body = $body;
    if (is_array($headers)) {
        if (sizeof($headers)>1)
            $headers=join(CRLF, $headers);
        else
            $headers=$headers[0];
    }
    if ($from) {
        $headers = preg_replace("!(from:\ ?.+?[\r\n]?\b)!i", '', $headers);
    }
    $this->headers = chop($headers);
    $this->mimeparts[] = "" ;    //Bump up location 0;
    return;
 }

?>
```

We have the constructor for our object and this takes as arguments the "from" and "to" email addresses, subject and body of the email and headers. For the body part, you can give the regular email you *would probably* type in. The last argument is optional (user-defined) headers. E.g., X-Mailer: MYMailer_1.0. Please note that `$headers` can either be an array of different headers you may want to send to the mailer or just a single container for one specific header. You can not send a From: header in the `$headers` argument; if it is found, that part is automatically stripped. You can use the multiple headers as follows: `array("X-Mailer: MYMailer_1.0", "X-Organization: PHPBuilder.com").`
`$mimeparts` is created with an empty slot 1 (index 0), the rationale for which we will see later.

Heart of it All: The Methods

We modularize the creation of the MIME message headers, MIME part headers and the creation of the final email message into separate methods. The implementation of the methods follows directly from the MIME basics we encountered earlier.

```
<?php

function attach($data, $description = "", $contenttype = OCTET, $encoding = BASE64, $disp = '') {
    if (empty($data))
        return 0;
    if (trim($contenttype) == '')
        $contenttype = OCTET ;
    if (trim($encoding) == '')
        $encoding = BASE64;
    if ($encoding == BIT7)
        $emsg = $data;
    elseif ($encoding == QP)
        $emsg = $$this->qp_func($data);
    elseif ($encoding == BASE64) {
        if (!$this->base64_func)     # Check if there is user-defined function
            $emsg = base64_encode($data);
        else
            $emsg = $$this->base64_func($data);
    }
    $emsg = chunk_split($emsg);
    //Check if content-type is text/plain and if charset is not specified append default CHARSET
    if (preg_match("!^".TEXT."!i", $contenttype) && !preg_match("!;charset=!i", $contenttype))
        $contenttype .= ";\r\n\tcharset=".CHARSET ;
    $msg = sprintf("Content-Type: %sContent-Transfer-Encoding: %s%s%s%s",
    $contenttype.CRLF,
    $encoding.CRLF,
    ((($description) && (BODY != $description))?"Content-Description: $description".CRLF:""),
    ($disp?"Content-Disposition: $disp".CRLF:""),
    CRLF.$emsg.CRLF);
    BODY==$description? $this->mimeparts[0] = $msg: $this->mimeparts[] = $msg ;
    return sizeof($this->mimeparts);
}

?>
```

Let us go over this method closely (as we will most of the other methods):

1. The method takes as arguments, the:
    - actual data to attach (`$data`).
    - description the data corresponding to the Content-Description header (`$description`).
    - content-type of the data as will be used in the Content-Type header (`$contenttype`).
    - encoding to be used and the value of Content-Transfer-Encoding (`$encoding`).
    - disposition value that can be used in the Content-Disposition header `$disp` can be INLINE or ATTACH, both of which are constants.
2. Values such as BASE64, TEXT, etc., are defined as constants in the accompanying .def file.
3. We use the `$encoding` value to decide what kind of encoding we need to do on the data. The valid values are BIT7 (for 7bit), QP or BASE64. This method also checks to see if the user wants to use his/her own BASE64 or QP functions. As of this writing, only BIT7 and BASE64 are implemented in our class, however, you can always pass your own quoted-printable function to use, via the `$qp_func` class variable discussed earlier.
4. After the encoding process, you will notice the use of `chunk_split()` on the encoded information. This function splits the string into smaller chunks for an optional length. Since we do not specify the length, the default length of 76 is used. This is very much in compliance with emailing conventions.
5. Next, if `$contenttype` argument contains `text/plain`, a "charset=" parameter value *must* be given. The value is defined in the constant CHARSET and is `us-ascii`. Note that when headers are passed with parameter values, there must be a delimiting semi-colon (;) between the header and parameter. E.g., `Content-Type: text/plain; charset=us-ascii`.
6. The other MIME part headers are created if their respective values are passed to this method. We do not want to have a Content-Description header without a description, after all. After creating the headers, we append the encoded data the part information. (Inspect the `sprintf()` statement in the method).
   Also, note that we use a special description field called BODY (again a constant). This is something we are using in our class implementation. If the description field is same as BODY, we use the first slot (index 0) of `$mimeheaders` array to attach. Read on for more on this.
7. `attach()` returns the current size of the $mimeparts array for the calling script's reference. That way one knows in which slot an attachment "X" is present (actually return value less one is the actual index in the array)
8. Note that all headers *must* be terminated with a CRLF (`\r\n`) sequence.

Next, we look at the method `fattach()`, `fattach()` is similar to `attach()`, but it takes a filename as its first argument (instead if `$data` of `attach()`). This method is just a wrapper so that the caller can invoke `fattach` with the filename. `fattach()` will then read in the file and call `attach()` to actually attach the data. This method returns a 0 on failure, an explanation for which can be found in the `$errstr` variable or on success, returns the slot number (less one) of the file attachment in the `$mimeparts` array.
We now have developed the capability to attach data, encode them and stow away the individual MIME parts in a private array. The jobs remaining to be done are:

- to complete the MIME parts
- to create the message headers with the MIME message headers included, the message original headers (as in To:, From: etc.) and include any user-defined headers.
- to append the completed MIME parts to the headers so that a complete email packet is generated.

The next method we will be examining, `build_message()`, does the bulk of all this work but is invoked via one `gen_email()` method. Please note that `build_message()` is a private method.

```
<?php

function build_message() {

    $msg = "";
    $boundary = 'PM'.chr(rand(65, 91)).'------'.md5(uniqid(rand()));    # Boundary marker
    $nparts = sizeof($this->mimeparts);

 // Case 1: Attachment list is there.  Therefore MIME Message header must have multipart/mixed
     if (is_array($this->mimeparts) && ($nparts > 1)) {
        $c_ver = "MIME-Version: 1.0".CRLF;
        $c_type = 'Content-Type: multipart/mixed;'.CRLF."\tboundary=\"$boundary\"".CRLF;
        $c_enc = "Content-Transfer-Encoding: ".BIT7.CRLF;
        $c_desc = $c_desc?"Content-Description: $c_desc".CRLF:"";
        $warning = CRLF.WARNING.CRLF.CRLF ;

 // Since we are here, it means we do have attachments => body must become an attachment too.
        if (!empty($this->body)) {
            $this->attach($this->body, BODY, TEXT, BIT7);
        }

 // Now create the MIME parts of the email!
        for ($i=0 ; $i < $nparts; $i++) {
            if (!empty($this->mimeparts[$i]))
                $msg .= CRLF.'--'.$boundary.CRLF.$this->mimeparts[$i].CRLF;
        }
        $msg .= '--'.$boundary.'--'.CRLF;
        $msg = $c_ver.$c_type.$c_enc.$c_desc.$warning.$msg;
    } else {
        if (!empty($this->body)) $msg .= $this->body.CRLF.CRLF;
    }
    return $msg;
}

?>
```

This method is kind of a paradox, it is simple yet complex. Read on to see for yourself.

1. We read earlier (under MIME Basics), that each MIME part has a boundary marker and that the marker is a unique id. The boundary marker is used in:
   - the MIME Message header to denote where the attachments must be demarcated
   - the MIME parts: actually before and after each part to delimit that attachment boundary. (Refer back to Alexander's email with the image and pedigree chart!)

   (Remember: The last boundary marker ends in two hyphens (--) to denote end of scope). `$boundary` contains the boundary marker and is the MD5 hash of the unique id of a random number. Additionally, we also prefix a "PM?" to `$boundary`, where "?" is a random alphabet. An example value for `$boundary` is `"PMK------2345ee5de0052eba4daf47287953d37e"`*(PM stands for PHP MIME, so you can change this to your initials may be!)*
2. We must consider two cases during the process of generation MIME headers. These cases affect the way in which the original body of the mail (`$body` in the constructor) is treated and the very presence of MIME headers.
   - Case 1 is the reason why this article was written and you are reading it: There are attachments available to be included! In this case, please note that what would have been the body of the message is occupied by the warning string `"This is a MIME Encoded Message"`. Hence, the actual message body itself must be added as an attachment to this message! The email text usually is the first attachment in the list of attachments, which is `$mimeparts[0]` in our case. That is precisely the reason why we bumped up an index in `$mimeparts` so that the first slot (subscript 0) is available for the email text part. The email body must be attached as text/plain with 7bit encoding.

     ```
     <?php

     if (!empty($this->body)) {
         $this->attach($this->body, BODY, TEXT, BIT7);
     }

     ?>
     ```
     The above snippet does this job of attaching the email text part as a MIME attachment. Note that we are using the 'BODY' constant to indicate to `attach()` *where* to attach it.
   - Case 2 is when there are no attachment, in which the case, the email text, if supplied, is the only information that is included in the local variable `$msg`; no MIME headers are needed in this case. (However, we could have specified just the MIME-Version header in this case - rewind back to the simplest MIME message presented earlier.)
3. The MIME message headers (MIME-Version, Content-Type, etc.) are created if there are attachments. To create the message body with the MIME Message headers, first the MIME message headers are created. Then the individual MIME parts available through the `$mimeheaders` array are processed in iteration. This is the point where the boundary marker (`$boundary`) is actually used. In conformance with the rules, two hyphens are prefixed (`>'--'.$boundary.CRLF`) for a MIME part and additionally, two hyphens are appended to the boundary marker (`'--'.$boundary.'--'.CRLF;`) after the last MIME part to indicate the end-of-scope.
4. The completed message in the variable `$msg` is the return value of this method.

The next method, `gen_email()` completes (well, more or less) the MIME message created by the `build_message()` method. Since `build_message()` is an internal method, `gen_email ()` creates the RFC 822 headers and appends the MIME information after a call to `build_message()`.

```
<?php

function gen_email($force=false) {

if (!empty($this->email) && !$force) return $this->email ;  // saves processing
    $email = "";
```

```
    if (empty($this->subject)) $this->subject = NOSUBJECT;
    if (!empty($this->from)) $email .= 'From: '.$this->from.CRLF;
    if (!empty($this->headers)) $email .= $this->headers.CRLF;
    $email .= $this->build_message();
    $this->email = $email;
    return $this->email;
}

?>
```

The class member `$email` has the entire email message generated for an instance of our class. To avoid unnecessary recreation of the message, this method proceeds to create the email headers and to call `build_message()` only if `$email` is empty. However, you can force reprocessing by calling `gen_email(true)`. (The caller will definitely want to do this if the "To" information is changed or a new attachment is added).
`gen_email()` creates the more familiar From header. Additionally it sets the subject to a generic (No Subject) if no subject was specified at all. We save the inclusion of the To and Subject headers until later. The method returns the completed email message and this concludes the task of creating the MIME message.
Two other methods that are worth mentioning are `print_mail()` and `send_mail()`, both taking the `$force` argument. `print_mail()` prints the entire email message and `send_mail()` sends the message using the PHP `mail()` function. Optionally, `send_mail()` uses an SMTP object and it's send method (both user specified) to send the email.

Conclusion

Creating MIME compliant messages is not as complicated as it first appears to be and can be implemented in a rather simple fashion. MIME messages can give a real facelift to many web site features.
The class we developed above covers core ideas and it can be extended in ways limited only by one's imagination. For instance, one can write a "detach()" function to remove an attachment given the slot number of an attachment (`attach()` method returns this information).
This class MIME_mail can be used to send HTML-based email without much alteration but embedded images can not be sent; this is a topic that needs special attention.
However, HTML without images or with images referenced using absolute URLs or `<BASE>` tags can be sent even using class MIME_mail. As an example:

```
<?php

$html_data =  '<html><body text="#0000dd" bgcolor="#000000"><h1>Hello</h1><body></html>';
$mime = new MIME_mail($to, $from, $subject);
$mime->attach($html_data,  "", OCTET, BASE64, INLINE);
$mime->send_mail ();

?>
```

The recipient of this email will get an email with a black background and the text "Hello" written in blue!
Sending complete inline HTML messages, along with some other advanced topics in MIME-compliant emailing, deserve special consideration, which, hopefully, will be addressed in a continuation of this article.
The class, constants file and working examples can be downloaded as a zip archive.
--Kartic