Spam, Wonderful Spam

Doug Hennig

This month's article puts emailing on steroids. The tools presented here provide the ability to send bulk emails to anyone listed in a table.

Last month, we looked at SFMAPI, a class that encapsulates the MAPI ActiveX controls that come with VFP. This class makes it easy to email-enable an application; you simply drop an SFMAPI object on a form, set some properties, specify recipients and attachments, and call the Send method to send a message. This month, we'll look at some reusable tools that put emailing on steroids: they provide the basis for bulk emailing. Despite the provocative title for this article <g>, I'm not advocating spam; there are lots of legitimate reasons to send the same message to a group of people. For example, you may want to notify customers when a new version of a product is available, let friends and family know your new address when you move, etc.

There are a few components to the bulk mailing tool presented in this article: a mail table, a bulk mailing class, an email class (like SFMAPI), and a form or other object that coordinates things.

Mail Table

In this article, I refer to the table containing the addresses to email to in a bulk emailing the "mail table". It could be a customer table that has an email address field, a cursor created from a SQL SELECT or view, etc. The name of the field containing the email address is unimportant, since you'll specify the name in a property of the bulk mailing object we'll see in a moment. The mailing object supports filtering this table, since you often don't want to send a message to every record.

I like to keep track of which addresses were actually processed, so I include a "mail sent" field in the table. If such a field doesn't exist in your table, you could add one by performing a SQL SELECT using code like this:

```
select *, .T. as MAILSENT from CUSTOMERS into cursor MAILTABLE
```

As with the email address, you'll specify the name of this field in a property of the bulk mailing object. If this field exists, it will be set to .T. as a record is processed; if not, this step is simply ignored. One advantage for having such a field is if the bulk mailing process is stopped and started again for some reason; the bulk mailing object has some logic that allows it to start from where it left off if this field exists.

SFMailer

SFMailer (SFMAILER.VCX) is the bulk mailing class. It's based on SFTimer, our timer base class in SFCTRLS.VCX. Why a timer? So emails can be sent as a background process. This allows the process to easily be stopped and started, which would be more difficult if it was running as a hard loop. The Enabled property of SFMailer is set to .F. so it doesn't begin processing until told to, and the Interval is set to 1000 (one second).

SFMailer doesn't do the actual sending of a message; it uses an email object, referenced in its oMail property, to do the dirty work. An example of such an object is the SFMAPI class we looked at last month. Because different email objects have different interfaces (for example, the name of the property containing the email address), you have to subclass SFMailer to use a particular email object.

Because SFMailer may be used in an interactive environment, it has the ability to notify another object (such as a form) when its status changes, such as when another email has been sent. Store a reference to the "subscribing" object in the oNotify property. This object must have a UpdateStatus method, because SFMailer's UpdateStatus method expects to call it.

Table 1 shows the custom properties in SFMailer.

Property	Purpose
*cAddress	The current email address
cEmailAddressField	The unaliased name of the field containing the email address (the default is "email")

cErrorMessage	The error message if something went wrong (an assign method for this property makes it read-only to anything outside this class)
cFilter	The filter to apply to the mail table
cMailAlias	The alias of the table to do the emailing from
cMessage	The message contents
cSentFlagField	The unaliased name of the “mail sent” flag in the mail table (the default is “mailsent”; an access method for this property returns a blank if an invalid field name was specified)
cSubject	The message subject
lProcessing	.T. if we’re currently sending emails (an assign method for this property makes it read-only to anything outside this class)
lResetSentFlag	.T. to reset the “mail sent” flag in all records before starting
nProcessed	The number of records processed (an assign method for this property makes it read-only to anything outside this class)
nRecords	The number of records to process (an assign method for this property makes it read-only to anything outside this class)
*oMail	An reference to the mail object
oNotify	A reference to an object to be notified about status changes

Table 1. Custom properties of SFMailer (indicates protected).*

The CountRecords method can be called once the cMailAlias, cFilter, cSentFlagField, and lResetSentFlag properties have been assigned the desired values. This method sets the nRecords property to the number of records in the mail table that meet the filter conditions specified in cFilter, and sets the nProcessed property to the number of records that have already been processed (if lResetSentFlag is .T., this is 0; otherwise, the number of those records that have the field specified in cSentFlagField set to .T.). This method calls the UpdateStatus method to notify the subscriber that the status has changed so, for example, a form can display the number of records to process.

```
local lnSelect, ;
  lcFilter
with This

* Ensure we have a mail table.

  do case
    case empty(.cMailAlias) or ;
      vartype(.cMailAlias) <> 'C'
      .cErrorMessage = 'cMailAlias not specified'
      return .F.
    case not used(.cMailAlias)
      .cErrorMessage = .cMailAlias + ' is not open'
      return .F.
  endcase

* Select the mail table and clear the filter.

  lnSelect = select()
  select (.cMailAlias)
  set filter to

* Count the total records matching the filter.

  lcFilter = iif(empty(.cFilter), '.T.', ;
    alltrim(.cFilter))
  count for &lcFilter to .nRecords

* Count the records that've already been processed.

  if .lResetSentFlag or empty(.cSentFlagField)
    .nProcessed = 0
  else
    lcFilter = iif(empty(.cFilter), '', ;
      '(' + alltrim(.cFilter) + ') and ') + ;
      .cSentFlagField
    count for &lcFilter to .nProcessed
  endif .lResetSentFlag ...
```

```
  .UpdateStatus()

* Reselect the former workarea.

  select (lnSelect)
endwith
```

The StartMail method starts the mail process. It first ensures the table specified in the cMailAlias property is open and that the cEmailAddressField property contains a valid field name in that table. It calls the GetMailObject method (abstract since the mechanism of getting the object will vary depending on the type of email object used) to ensure we have an object that'll handle the actual emailing for us, and flags all records as having not been sent if the lResetSentFlag property is .T. It then sets a filter on the mail table and moves to the first record matching the filter. Before returning, it sets the lProcessing flag to .T., calls the UpdateStatus method to notify any object that the status has changed, and starts the timer.

```
local lnSelect, ;
  lcSentFlag, ;
  lcFilter
with This

* Ensure everything is set up correctly.

  do case
    case empty(.cMailAlias) or ;
      vartype(.cMailAlias) <> 'C'
      .cErrorMessage = 'cMailAlias not specified'
      return .F.
    case not used(.cMailAlias)
      .cErrorMessage = .cMailAlias + ' is not open'
      return .F.
    case empty(.cEmailAddressField) or ;
      not type(.cMailAlias + '.' + ;
      .cEmailAddressField) $ 'CM'
      .cErrorMessage = 'cEmailAddressField not ' + ;
        'properly specified'
      return .F.
  endcase

* Ensure we have a mail object.

  .GetMailObject()
  if vartype(.oMail) <> 'O'
    .cErrorMessage = 'Could not create mail object.'
    return .F.
  endif vartype(.oMail) <> 'O'

* Select the mail table.

  lnSelect = select()
  select (.cMailAlias)

* Set the "mail sent" flag to .F. if necessary.

  lcSentFlag = .cSentFlagField
  if .lResetSentFlag and not empty(lcSentFlag)
    replace all &lcSentFlag with .F.
  endif .lResetSentFlag ...

* Set a filter and go to the first record.

  lcFilter = .cFilter
  if not empty(lcSentFlag)
    lcFilter = iif(empty(lcFilter), '', '(' + ;
      lcFilter + ') and ') + 'not ' + lcSentFlag
  endif not empty(lcSentFlag)
  set filter to &lcFilter
  locate
```

```
* Flag that we're processing, update the status,
* reselect the former workarea, and start the timer.

  .lProcessing = .T.
  .UpdateStatus()
  select (lnSelect)
  .Enabled = .T.
endwith
```

Once the timer is enabled, the Timer method executes when the timer fires. This method calls the SendMail method if lProcessing is .T. or disables the timer if not.

```
with This

* If we're processing emails, send the next message.
* Otherwise, stop the timer.

  if .lProcessing
    .SendMail()
  else
    .Enabled = .F.
  endif .lProcessing
endwith
```

The SendMail method processes the current record in the mail table. It gets the email address from the field specified in the cEmailAddressField and puts it into the cAddress property. If it isn't empty, the timer is disabled so it doesn't fire again while we're processing this record, and the abstract SendMessage method is called (the reason this method is abstract is because the actual mechanism to send a message will vary with the mail object we're using). The field specified in the cSentFlagField property is set to .T. (indicating that a message was sent to this address), the nProcessed property is incremented, and the UpdateStatus method is called to notify any object. Finally, the record pointer is moved to the next record, and if we're at the end of the table, the StopMail method is called; otherwise, the timer is re-enabled.

```
local lcSentFlag
with This

* Select the mail table.

  lnSelect = select()
  select (.cMailAlias)

* Get the email address for the current record. If it
* isn't empty, let's process it.

  .cAddress = alltrim(evaluate(.cEmailAddressField))
  if not empty(.cAddress)

* Disable the timer for this process and have the mail
* object send the message.

    .Enabled = .F.
    .SendMessage()
  endif not empty(.cAddress)

* Update the "mail sent" flag.

  lcSentFlag = .cSentFlagField
  if not empty(lcSentFlag)
    replace &lcSentFlag with .T.
  endif not empty(lcSentFlag)

* Increment the number of records processed and update
* the status.

  .nProcessed = .nProcessed + 1
  .UpdateStatus()
```

```
* Move to the next record. If we're at EOF, we're done.
* Otherwise, reenable the timer.

  if not eof()
    skip
  endif not eof()
  select (lnSelect)
  if eof(.cMailAlias)
    .StopMail()
  else
    .Enabled = .T.
  endif eof(.cMailAlias)
endwith
```

The StopMail method simply terminates the processing by disabling the timer and setting the lProcessing property to .F. It also calls UpdateStatus to notify any object that the status has changed. This method is called from SendMail when the end of the mail table is reached, but could also be called from something else that wants to stop the process.

SFMailerMAPI

I created a subclass of SFMailer called SFMailerMAPI that uses an SFMAPI object to send a message. SFMailerMAPI overrides two methods: GetMailObject and SendMessage. GetMailObject grabs the SFMAPI object attached to the form it's on if necessary and possible.

```
with This
  if vartype(.oMail) <> 'O' and ;
    type('Thisform.Name') = 'C'
    .oMail = Thisform.oMAPI
  endif vartype(.oMail) <> 'O' ...
endwith
```

Of course, if you haven't dropped the SFMailerMAPI object on a form or if that form doesn't have an SFMAPI object called oMAPI, you'll have to set the oMail property differently.

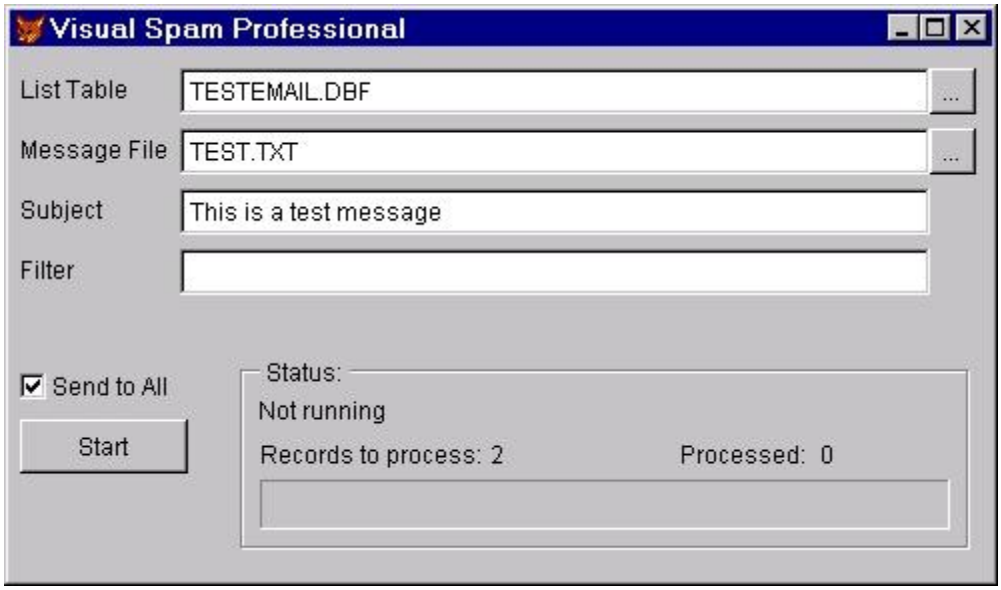
SendMessage uses the necessary properties and methods of the SFMAPI object to send a message (see last month's article for details).

```
with This.oMail
  .NewMessage()
  .cSubject = alltrim(This.cSubject)
  .cMessage = alltrim(This.cMessage)
  .AddRecipient(alltrim(This.cAddress))
  .Send()
endwith
```

Using SFMailer

VSPRO.SCX (which stands for "Visual Spam Professional"; thanks to Mike Feltman for suggesting the name <g>) shows how to use SFMailer. As you can see, there are textboxes for the name of the mailer table, the name of a file containing the body of the message (you could also provide an editbox and type the message directly into the form), the subject of the message, and a filter to apply to the table. The "Send to All" checkbox is bound to Thisform.oMailer.lResetSentFlag, so the user can indicate whether it should process records already flagged as sent (the default when the form is started) or start from where it left off last time (the default after the process is stopped). The status area shows the number of records to process and the number processed (which come from the nRecords and nProcessed properties of the mailer object), and a ActiveX ProgressBar control shows the progress as it's running. The Start button starts the process; its Caption becomes "Stop" while the process is running.

Figure 1. VSPRO.SCX.



This form has cMailerClass and cMailerLibrary properties which specify which class to instantiate as the mailer object; I've set them to SFMailerMAPI and SFMailer.vcx. The Load method of the form instantiates the class specified in these properties as oMailer; this is done in Load rather than Init because several controls on the form are bound to properties of oMailer. Load also sets the oNotify property of the mailer object to This; the form's UpdateStatus method will query various properties of the mailer object and display the status of the process (including setting the Caption of the Start button to "Stop" if the email process is currently running). The form also has an SFMAPI object on it (with the name oMAPI) so the SFMailerMAPI object can use it for sending messages.

If the emailing process isn't running, the Click method of the Start button reads the contents of the specified message file into the cMessage property of the mailer object, then calls its StartMail method. If the process is running, the StopMail method is called.

```
with Thisform
  if This.Caption = 'Start'
    .oMailer.cMessage = filetostr(.txtMessageFile.Value)
    .oMailer.StartMail()
  else
    .oMailer.StopMail()
    .oMailer.lResetSentFlag = .F.
  endif This.Caption = 'Start'
endwith
```

Other methods in the form are really just "glue" code to ensure that everything is set up correctly before the process can start.

SFMailerwwIPStuff

There are at least three drawbacks to using MAPI to send bulk emails: they're send synchronously (so you have to wait for each message to be sent before the next one can be processed), each message appears in the Sent Folder of your email application (which you may think of as an advantage, depending on your needs) and, at least when you use Outlook Express as I do, you get a status dialog for each message. I prefer to use wwIPStuff, a shareware set of utilities from West Wind Technologies (see www.west-wind.com for details and a download file), because it avoids all three of these issues.

I created a subclass of SFMailer called SFMailerwwIPStuff that uses wwIPStuff to send messages. I added cMailServer, cSenderEmail, cSenderName, and cwwIPStuffDirectory properties (the first three contain information about your mail server and you, which aren't need in SFMailerMAPI since they're specified in your mail client, and the last specifies the directory where wwIPStuff is installed). The

GetMailObject method instantiates a wwIPStuff object into the oMail property, and the SendMessage method sets the appropriate properties of oMail and calls its SendMailAsync method to send the message.

Conclusion

Although I joked about spam a couple of times in this article, I hope you'll use these classes for legitimate purposes, such as client contact or employee e-newsletters. Don't call me if your ISP shuts you down!

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author or co-author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit, Stonefield Query, and Stonefield Reports. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at the 1997, 1998, and 1999 Microsoft FoxPro Developers Conferences (DevCon) as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Microsoft Certified Professional (MCP). He can be reached at dhennig@stonefield.com.