Get the Message?

Doug Hennig

This month's article presents a class that can be used to send email from your applications. It combines the two MAPI ActiveX controls that come with VFP into one easy-to-use class.

More and more users expect their applications to be email-enabled these days. For example, the customer form of a typical business application may have a field to store the customer's email address. It'd be a shame not to have a quick way (such as a button on the form or in a toolbar, or a menu item) to send that customer an email without having to switch to something like Outlook Express and copy and paste their email address. It's doubly a shame since it's so easy to add that capability to VFP applications. This month's article looks at some classes that can be used to send emails within your applications.

MAPI ActiveX Controls

The simplest way to email-enable an application is to use the MAPI ActiveX controls that come with VFP. There are two related controls, both contained in MSMAPI32.OCX: MAPIMessages and MAPISession. These controls work together; the MAPISession control is responsible for managing the mail session and the MAPIMessages control is used to send and receive messages. Both of these controls are non-visual. You can either drop them on a form or instantiate them in code with CREATEOBJECT (the ProgIDs for these controls are MSMAPI.MAPIMessages and MSMAPI.MAPISession).

The properties, events, and methods (PEMs) of these controls are described in the help file, MAPI98.CHM (located in \WINDOWS\HELP on my system). Rather than describing how to use these controls directly, we'll look at a container class I've built that combines these two controls into one easy-to-use object. We'll see how the PEMs of the ActiveX controls are used as we examine the code in the class. Also, although these controls can both send and receive email, this article will just focus on sending messages.

SFMAPI

To hide the complexity of working with two controls, I created the SFMAPI class (in SFMAPI.VCX). SFMAPI is based on SFContainer, our base container class defined in SFCTRLS.VCX. It contains a MAPIMessages control (named oMAPIMessage in SFMAPI) and a MAPISession control (oMAPISession). I set the Visible property of the container to .F. (since it's a non-visual control at runtime) and added several properties: aAttachments, a protected array of file names to attach to a message; aRecipients, a protected array of recipients for a message; cMessage, the body of the message to send; cSubject, the subject of the message; lLoggedIn, a protected property which contains .T. if we've successfully logged into MAPI; and lSignoffAfterSend, which if .T. means SFMAPI should log off the MAPI session after sending a message.

To send a message using SFMAPI, start by calling the NewMessage method. This message simply initializes aRecipents and aAttachments to single empty entries; it's really only needed if you previously sent a message with the object. Next, call the AddRecipient method once for each recipient of the message, passing the email address and optionally the name and type of the recipient (1 for a primary recipient, 2 for a copy recipient, and 3 for a blind copy recipient). This method adds the passed information to the aRecipients array after ensuring it's valid:

```
lparameters tcAddress, ;
  tcName, ;
  tnType
local lcName, ;
  lnType, ;
  lnCount
assert vartype(tcAddress) = 'C' and ;
  not empty(tcAddress) ;
  message 'Invalid address specified'
lcName = iif(type('tcName') <> 'C' or empty(tcName), ;
  tcAddress, tcName)
lnType = iif(type('tnType') <> 'N' or ;
  not between(tnType, 1, 3), 1, tnType)
```

```
with This
  lnCount = iif(empty(.aRecipients[1, 1]), 1, ;
    alen(.aRecipients, 1) + 1)
  dimension .aRecipients[lnCount, alen(.aRecipients, 2)]
  .aRecipients[lnCount, 1] = lcName
  .aRecipients[lnCount, 2] = tcAddress
  .aRecipients[lnCount, 3] = lnType
endwith
```

If you have any attachments for the message, call the AddAttachment method, which is similar to AddRecipient but adds the specified filename to the aAttachments array.

Finally, set the cSubject property to the subject of the message, set cMessage to the body of the message, and call the Send method. Pass .T. if you want a dialog box displayed; note that you must do this when calling this method from VFP 5 to avoid getting an "invalid callee" error (it works correctly in VFP 6).

We'll look at the Send method in pieces. It begins by ensuring we have at least one recipient, then calling the SignOn method (which we'll look at shortly) if we're not already logged into MAPI. If it fails, Send returns .F.

```
lparameters tlDisplayDialog
local lnI
with This

* Ensure that we have at least one recipient.

  assert not empty(.aRecipients[1, 1]) ;
    message 'SFMAPI.Send: no recipients'

* If we're not already logged in, try to do so. If we
* can't, return .F.

  if not .lLoggedIn and not .SignOn()
    return .F.
  endif not .lLoggedIn ...
```

Next, it sets the MAPIMessages object's SessionID property to the value of the MAPISession object's property so the MAPIMessages object can communicate with MAPI; this is similar to using a file handle for a file opened with the FOPEN() function. It calls the MAPIMessages Compose method to start a new message, and sets the MsgNoteText and MsgSubject properties to its cMessage and cSubject values.

```
.oMAPIMessage.SessionID = .oMAPISession.SessionID
.oMAPIMessage.Compose()
.oMAPIMessage.MsgNoteText = .cMessage
.oMAPIMessage.MsgSubject  = .cSubject
```

The next step is to tell the MAPIMessages object about the recipients. The MAPIMessages object has an internal collection of recipients that's exposed in kind of a weird way: the recipient-related properties are bound to the current internal recipient, which is indicated by the RecipIndex property. You set RecipIndex to the index of the recipient you want to work with (the index is zero-based, so the first recipient has an index of 0), and the appropriate MAPIMessages properties read from and write to the selected recipient. To add a new recipient, set RecipIndex to a value greater than the current number of recipients (actually, equal to the number, since the index is zero-based), which is stored in the RecipCount property; doing so automatically increments RecipCount. The recipient-related properties we're interested in are RecipDisplayName (the name of the recipient as it's displayed to the user), RecipAddress (the email address), and RecipType (the recipient type, which has the range of values I discussed earlier). After setting these properties, call the ResolveName method to resolve the name from the address book as necessary.

The code to write the recipient information to the MAPIMessages object spins through the aRecipients array, setting the recipient index to add another recipient and updating the appropriate properties for the new recipient.

```
for lnI = 1 to alen(.aRecipients, 1)
  .oMAPIMessage.RecipIndex       = ;
```

```
    .oMAPIMessage.RecipCount
  .oMAPIMessage.RecipDisplayName = .aRecipients[lnI, 1]
  .oMAPIMessage.RecipAddress     = .aRecipients[lnI, 2]
  .oMAPIMessage.RecipType        = .aRecipients[lnI, 3]
  .oMAPIMessage.ResolveName()
next lnI
```

File attachments are treated very similarly: the AttachmentIndex property indicates which internal attachment is being referred to, and the AttachmentPosition, AttachmentPathName, and AttachmentName properties are bound to the current attachment. AttachmentPathName (the complete path and name for the attachment) and AttachmentName (the name of the attachment as the recipient sees it) are straightforward, but AttachmentPosition is (once again) weird: it indicates where in the body of the message the attachment should go. I'm not sure why attachments have anything to do with the body of the message, but since no two attachments can appear at the same position and none can be placed beyond the end of the message, I decided to place them at the front of the message (the position is zero-based; hence the –1 in the code below).

```
for lnI = 1 to alen(.aAttachments)
  if not empty(.aAttachments[lnI])
    .oMAPIMessage.AttachmentIndex    = ;
      .oMAPIMessage.AttachmentCount
    .oMAPIMessage.AttachmentPosition = lnI - 1
    .oMAPIMessage.AttachmentPathName = .aAttachments[lnI]
    .oMAPIMessage.AttachmentName     = ;
      justfname(.aAttachments[lnI])
  endif not empty(.aAttachments[lnI])
next lnI
```

Finally, we need to send the message by calling the MAPIMessages object's Send method. If we're supposed to display a dialog (as I mentioned earlier, this is required in VFP 5), the value 1 is passed to Send. We then call the SignOff method if we're supposed to sign off after sending a message. You might think it would make sense to return the value of the MAPIMessage's MsgSent property (which indicates if the message was successfully sent to the mail server or not; this doesn't indicate whether the mail will be successfully sent from the server or not) but for some reason, this property is always .F. for me.

```
  if tlDisplayDialog
    .oMAPIMessage.Send(1)
  else
    .oMAPIMessage.Send()
  endif tlDisplayDialog
  if .lSignOffAfterSend
    .SignOff()
  endif .lSignOffAfterSend
endwith
```

The protected SignOn method, which is called from Send if we're not already logged into MAPI, is used to log on to MAPI. It first ensures that MAPI32.DLL can be found in the Windows System directory (the GetSystemDirectory API function is used to locate that directory), then sets the MAPISession object's DownloadMail property to .F. (we only want to send messages at this time) and the LogonUI property to .F. (set this to .T. if you need to display a logon dialog so the user can enter their name and password). It calls the MAPISession's SignOn method to perform the logon to MAPI, and then sets SFMAPI's lLoggedIn property to .T. if the MAPISession's SessionID property is greater than 0. The method returns .T. if the login succeeded.

```
local lcDirectory, ;
  lnLen
with This

* Don't try to log onto MAPI unless we can find the DLL.

  declare integer GetSystemDirectory in Win32API ;
    string @, integer
```

```
  lcDirectory = replicate(chr(0), 80)
  lnLen       = GetSystemDirectory(@lcDirectory, 80)
  lcDirectory = addbs(left(lcDirectory, lnLen))

* We can find it, so set some properties and try to
* log in.

  if file(lcDirectory + 'MAPI32.DLL')
    .oMAPISession.DownloadMail = .F.
    .oMAPISession.LogonUI      = .F.
    .oMAPISession.SignOn()
    .lLoggedIn = .oMAPISession.SessionID > 0
  else
    messagebox('Cannot find MAPI32.DLL')
    .lLoggedIn = .F.
  endif not file(lcDirectory + 'MAPI32.DLL')
endwith
return This.lLoggedIn
```

The protected SignOff method, which is called from Send if the lSignOffAfterSend property is .T., from the Destroy method, and from the Error method, logs off MAPI and sets the lLoggedIn property to .F.

```
with This.oMAPISession
  if .SessionID <> 0
    .SignOff()
  endif .SessionID <> 0
endwith
This.lLoggedIn = .F.
```

Sample Form

TESTMAPI.SCX is a form that demonstrates the use of SFMAPI. It looks like a typical customer data entry form, with textboxes for name, address, etc. It also has a textbox for the customer's email address. The label preceding this textbox is bolded (to show the user it's special) and has the following code in its DblClick method:

```
with Thisform.oMAPI
  .AddRecipient(alltrim(Thisform.txtEmail.Value))
  .Send(.T.)
endwith
```

When you double-click the label, this code adds the value of the email textbox as the recipient and calls the Send method of the SFMAPI control, passing .T. so a dialog is displayed. Notice the cSubject and cMessage properties weren't set; after all, we don't know what the user wants for these items, so they can enter the desired information in the email dialog that appears.

You could provide other ways for the user to indicate they want to send an email: a command button beside the email textbox, a button on the form or in a toolbar, a menu item, etc. Well-known VFP guru Markus Egger created an ingenious mechanism as part of Visual LandPro, an application he helped develop that was a finalist in the VFP Excellence Awards two years in a row. He created a textbox that displays an email address as a hyperlink (that is, it appears as blue underlined text on a white background). The user clicks on it, just as they would a hyperlink, to send a message to that address. The great thing about his class is that you don't even have to explain how it works to most users.

Conclusion

Since adding email capabilities to an application is as simple as dropping an SFMAPI object on a form, setting its properties, and calling some methods when the user indicates they want to send an email, there's no reason not to add these capabilities to even the simplest applications.

Next month, we'll look at some reusable tools that put emailing on steroids: they provide the basis for bulk emailing (no, I'm not advocating spam; there are lots of legitimate reasons to send the same message to a group of people). Until then, I hope you enjoy these classes.

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.