Reading the Mail

Click & Retrieve Source CODE!

BY PETER VOGEL

Use VB3, the MAPI VBX, and components of Microsoft Office to create a complete Workflow Automation application for both in-house and remote use.

Would you like to have your computer request that processing be done by another computer on the network? Would you like to be notified when jobs running on an unattended computer run to completion? How about getting "almost workflow" benefits with minimal changes to your Excel, Word, and Access applications? Maybe these features should come with built-in fault tolerance, auditing, administration, and logging capabilities. And of course, you want to get all these features for free. I did, too, and I didn't have to look any further than Microsoft's Mail API (MAPI) as implemented in VB's MAPI.VBX.

To take advantage of mail technology from within your VB application, you need to start with a generic program to read your mail (if e-mail terminology is new to you, look at the accompanying sidebar, "A Brief Introduction to Mail Systems").

Peter Vogel is the applications supervisor at Champion Road Machinery, a Microsoft Certified Solution Developer, the father of two terrific boys, and Jan's husband (not necessarily in that order). He can be reached at peter.vogel@odyssey.on.ca.

Let's call this program MailHandler (MAILHAND.EXE) and create it as a single form with two controls, MAPISESSION.VBX and MAPIMESSAGES.VBX.

MailHandler's code starts by logging onto the mail session of the computer it is running on:

```
Sub LogInMail
   'pick up any new mail on startup
   MAPISESSION.DownloadMail = True
   'piggyback on existing mail session
   MAPISESSION.NewSession = False
   'start session
   MAPISESSION.Action = 1
```

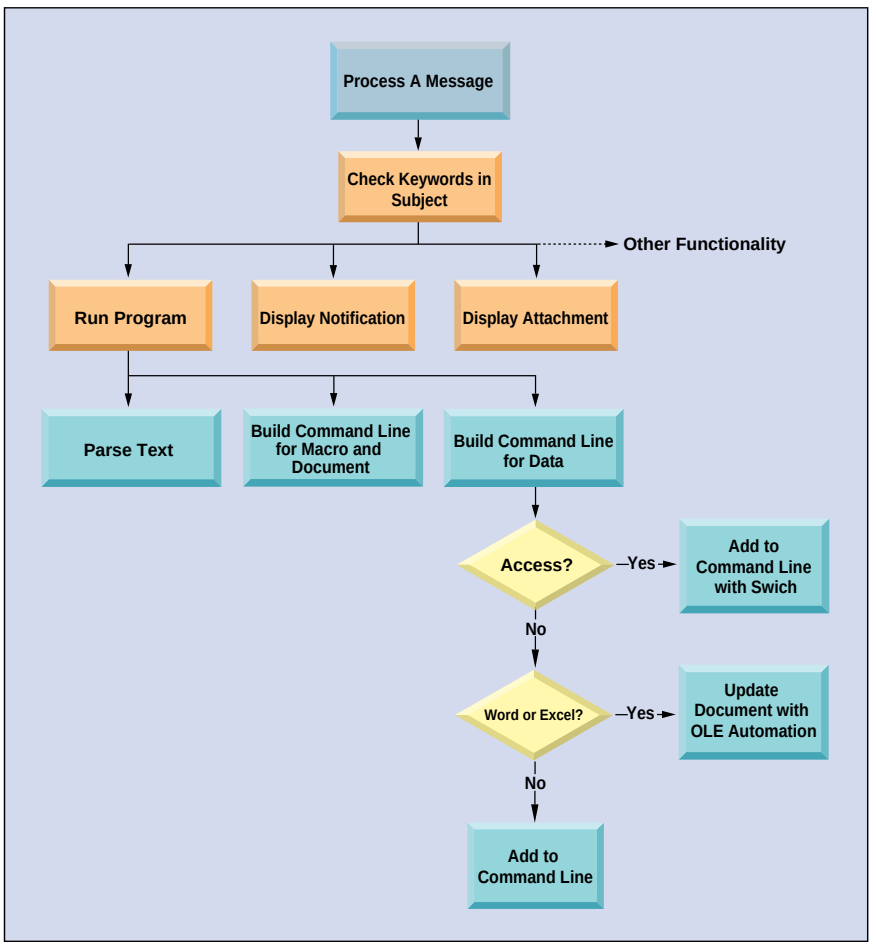

FIGURE 1 ***Mail Automation from Start to Finish.*** *Processing mail consists of logging onto the existing Mail session and repeatedly building lists of new mail. Each list must be processed, looking for messages to be handled by MailHandler. When a new mail list adds no new messages, processing ends.*

```
   'pass session ID to message VBX
   MAPIMESSAGES.SessionID = _
      MAPISESSION.SessionID
End Sub
```

Working with MAPISESSION establishes the mail session. All handling of the mail is done through MAPIMESSAGE, which is passed the ID for the session from MAPISESSION. This code assumes that the user has already logged onto the mail system. If your standard is to include mail in your users' startup groups, it'll work. Otherwise, you must set the logon information in your code:

```
MAPISESSION.UserName = "MailUser"
MAPISESSION.Password = "UserPassword"
```

Microsoft Mail gives you a third option: put UserId and Password information in the user's INI file for automatic logon at startup (an unattended computer at my office is configured in this manner). The last thing you need to do is shutdown the mail session you created using:

```
Sub LogOutMail
   MAPISESSION.Action = 2
End Sub
```

STRATEGY, FAILURE, RECOVERY

In my opinion, you have three usable strategies for processing mail. The first method requires that you start MailHandler in the StartUp Group and log on at startup, check for Mail in a Timer event and process any messages that require it, log out, and shut down.

In this scenario, the logon code goes in your form's Load event and the logout code goes in the form's Close event. This strategy consumes the most resources over the longest period of time, but it also has the smallest performance hit when mail is actually processed. You will need to add the timer control to your form in order to use this strategy.

To use the second method, you also place the MailHandler program in the StartUp Group, log onto mail in the Timer event, process mail, and log out. The logon and logout code go in the Timer event.

This consumes fewer resources over a shorter period of time, but it also causes a bigger memory hit when it's time to process mail. As with the first strategy, you'll need to add a timer control to your form.

The third method causes MailHandler to start automatically when new mail is received, log on, process mail, and log out. The logon code goes in the Load event and the logout code goes in the Close event. This method consumes the least amount of resources and does so the least frequently, but causes the biggest resource hit when it is time to process new mail.

Initially, I went with the first method for implementation at my company. However, at the time we implemented the system, we were having some problems with our network and this method turned out to be unreliable: I'd leave the computer alone for a while and return to find an "Unable to read from device NETWORK" message on the screen.

Converting to the third method from the first consisted of putting an End statement after the mail processing and a line in the MSMAIL.INI file. It seemed simple enough, so we went with it. Actually, we left both processes in with a command-

VB3

```
'define what the keywords to look for
'in the subject line of the messages
Global Const RUNPROGRAM = "RUN PROGRAM"
Global Const DISPLAYATTACHMENT = "DISPLAY ATTACHMENT"
Global Const DISPLAYNOTIFICATION = "NOTIFY MESSAGE"
Declare Function GetModuleUsage% Lib "Kernel" (ByVal hModule As Integer)
Declare Function FindExecutable% Lib "shell.dll" (ByVal lpszFile$, ByVal _
  lpsDir$, ByVal lpszResult$)
  Sub ProcessMail
'read all unread mail and respond to
'mail needing processing
Dim stzType As String
Dim stzId As String
Dim ing As Integer, ingUnRead As Integer

MAPIMessages.FetchUnreadOnly = True
MAPIMessages.Action = 1
ingUnRead = MAPIMessages.MsgCount
While ingUnRead > 0
  For ing = 0 To ingUnRead - 1
    MAPIMessages.MsgIndex = ing
    stzType = CheckType((MAPIMessages.MsgSubject))
     If stzType > " " Then
       ProcessMessage MAPIMessages, stzType, ing
       ingUnRead = ingUnRead - 1
     End If
  Next ing
  'check to see if any new mail recieved
  MAPIMessages.FetchUnreadOnly = True
  MAPIMessages.Action = 1
  If MAPIMessages.MsgCount > ingUnRead Then
    ingUnRead = MAPIMessages.MsgCount
  Else
    ingUnRead = 0
  End If
WendEnd ProcessMail
Function CheckType (stzSubject as string) as string
If Instr(UCase(stzSubject),RUNPROGRAM) Then
  CheckType =  RUNPROGRAM
ElseIf Instr(UCase(stzSubject),DISPLAYNOTIFICATION) Then
  CheckType =  DISPLAYNOTIFICATON
ElseIf Instr(UCase(stzSubject),DISPLAYATTACHMENT) Then
  CheckType = DISPLAYATTACHMENT
Else
  CheckType = " "
Endif
End Function
Sub ProcessMessage(cntMailControl as control ,stzType as string, ingPos as
  integer)
'passed the type of mail,
'call the routine to process it
Select Case (stzType)
  Case DISPLAYNOTIFATION
    DisplayNotification cntMailControl, ingPos
  Case RUNPROGRAM
    RunProgram cntMailControl, ingPos
  Case DISPLAYATTACHMENT
    DisplayAttachment cntMailControl, ingPos
  Case Else
    MsgBox "Mail Type " & stzType & " is recognized _
      but no routine exists to process it."
End Select
End If
```

LISTING 1 ***The Check is in the Mail?*** *The routine charged with checking for interesting mail, CheckType is called from within ProcessMail and returns the type of mail just read. This is passed to ProcessMessage, which finds the message and calls the appropriate routine to process it.*

VB3

```
Sub RunProgram (cntMail as control, ingPos as integer)
'passed a mail control, load the keywords
'array with the keyword to search for,
'get the name of the program and any parameters.
'check for special EXE's that require funny
'formatting of their command lines
Dim stzKeywords(4,2)
Dim stzCommandline as string
stzKeywords(1,1) = "Program:"
stzKeywords(2,1) = "Document:"
stzKeywords(3,1) = "Macro:"
stzKeywords(4,1) = "CommandLine:"
ParseKeywords stzKeywords(), cntMail.Text
If stzKeywords(1,2) > " " Then
  If stzKeywords(2,2) > " " Then
    stzCommandLine = stzKeywords(2,2)
  Endif
  If stzKeywords(3,2) > " " Then
    If Instr(stzKeywords(1,2), "MSAccess.Exe") > 0 _
      Then
      stzCommandLine = stzCommandLine  & "/X " & _
        stzKeywords(3,2)
    ElseIf Instr(stzKeywords(1,2), "WinWord.Exe") > _
      0 Then
      stzCommandLine = stzCommandLine  & "/m" & _
        stzKeywords(3,2)
    ElseIf Instr(stzKeywords(1,2), "Excel.Exe") > 0 _
      Then
      SetExeclMacro stzKeywords(3,2), _
        stzKeywords(2,2)
    Else
      stzCommandLine = stzCommandLine  & " " _
        & stzKeywords(3,2)
    End If
  Endif
  If stzKeywords(4,2) > " " Then
    If Instr(stzKeywords(1,2), "MSAccess.Exe") > 0 _
      Then
      stzCommandLine = stzCommandLine  & "/C " & _
        stzKeywords(4,2)
    ElseIf Instr(stzKeywords(1,2), "WinWord.Exe") > _
      0 Then
      SetWordCommand stzKeywords(4,2) ,_
        stzKeywords(2,2)
    ElseIf Instr(stzKeywords(1,2), "Excel.Exe") > 0 _
      Then
      SetExcelCommand stzKeywords(4,2),_
        stzKeywords(2,2)
    Else
      stzCommandLine = stzCommandLine  & " " _
        & stzKeywords(4,2)
    End If
  Endif
  WaitForCompletion Shell(stzKeywords(1,2) & " " _
    & stzCommandLine,4)
Endif
End Sub
Sub ParseText (stzKeywords() as string,stz as string)
' passed a 2-D array of  keywords and a
' string, add the value found for the
' keyword to the array
Dim stzWork as string
Dim ing as integer
stzWork = Trim(stz)
For ing = 0 to UBound(stzKeywords())
  ingWordPos = Instr(stzWork,stzKeywords(ing,1))
  If ingWordPos > 0 Then
    ingTabPos = Instr(ingWordPos,stzWork,Chr(9))
    ingCRPos = Instr(ingWordPos,stzWork,Chr(13))
    stzKeyWords(ing,2) = Mid$(stzWork,ingTabPos _
      + 1,ingCRPos-ingTabPos -1)
  End If
Next ing
End Sub
Sub WaitForCompletion(ingTaskId as Integer)
Sub RunProgram(cntMail as Control)
'passed a taskid, do not return until one of
'the instances of the task's module stop
Dim ingInitialUsage As Integer
If ingTaskId <> 0 Then
  ingInitialUsage = GetModuleUsage%(ingTaskId)
  If ingInitialUsage > 0 Then
    While GetModuleUsage%(ingTaskId) >= _
      ingInitialUsage
      DoEvents
    Wend
  End If
End If
End Sub
```

LISTING 2 ***Looking for Keywords.*** *The ParseText routine provides a standard way for each message-type routine to find its keywords. With VB4's improved variants, you can implement this as a function. RunProgram is the workhorse routine in MailHandler and shows how the ParseText routine is used. WaitForCompletion uses a Windows API call to pause MailHandler until the program completes.*

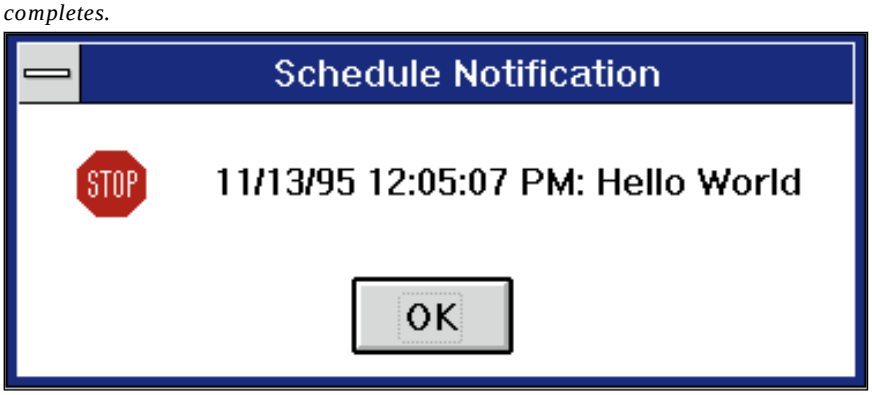

FIGURE 2 ***It's a First.*** *This is a standard first message for any new tool. The displayed date and time let the recipient know if the message had been received on time.*

line option to switch between them:

```
Sub Form_Load()
' check to see if MailHandler
' already running and exit if so.
If App.PrevInstance Then
   End
End If
LogInMail
If Command() = "/T"
   'strategy 1
   Timer.Enabled = True
Else
   'strategy 2
   Timer.Enabled = False
   ProcessMail
   End
End If
End Sub
Sub Timer_Timer()
   ProcessMail
End If
Sub UnLoad(Cancel as integer)
   LogOutMail
End Sub
```

To get MSMAIL to run MailHandler whenever it received new mail, we added this line to the MSMAIL.INI file on the computer that was to process the mail:

```
AUTO=3.0;;;;APPEXEC.DLL;C:MAILHAND.EXE;_
   0 010000000000000;_
   Process application system mail;;;
```

We also had to ensure that APPEXEC.DLL was in the Windows/System directory of the computer. APPEXEC is supposed to be a custom DLL that you create to pass parameters to the program called when new mail is received. Be-

User Tip VB4

UNREGISTER A DLL WITH THE RIGHT MOUSE BUTTON IN WIN95

Create a file called UNREGDLL.REG, and add this to the file:

```
REGEDIT4

[HKEY_CLASSES_ROOT\dllfile\shell]
@="open"

[HKEY_CLASSES_ROOT\dllfile\shell\open]
@=""

[HKEY_CLASSES_ROOT\dllfile\shell\open\command]
@="C:\\windows\\system\\regsvr32.exe %1"

[HKEY_CLASSES_ROOT\dllfile\shell\Unregister]

[HKEY_CLASSES_ROOT\dllfile\shell\Unregister\command]
@="C:\\windows\\system\\regsvr32.exe /u %1"
```

Run the file in Windows 95. Now, when you right-click on a DLL or OCX file in Explorer, the Unregister command will appear, allowing you to unregister a DLL quickly. You can delete the REG file after you run it the first time.

To register and unregister a DLL or OCX from the command line, use RegSvr to register a 16-bit DLL or OCX and use RegSvr32 to register a 32-bit DLL or OCX. To unregister a DLL, use the same command with the /u switch. For example:

```
RegSvr32 msrdo.dll              registers the RDO dll
RegSvr32 /u msrdo.dll           unregisters the RDO dll
```

—A. Nicklas Malik
received by e-mail

SEND YOUR TIP

If it's cool and we publish it, we'll pay you $25. If it includes code, limit code length to 10 lines if possible. Be sure to include a clear explanation of what it does and why it is useful. Send to 74774.305@compuserve.com or Fawcette Technical Publications, 209 Hamilton Ave., Palo Alto, CA, USA, 94301-2500.

cause MailHandler doesn't need any parameters, you can use APPEXEC, a dummy DLL supplied with MSMail.

Now you're ready to have MailHandler see what's in the mail. The trick is to recognize which messages can be processed automatically, and which messages are intended to be read by the user (see Listing 1).

Define three types of mail to be processed automatically: display a message, run a program, and display an attachment. All three types will make their appearances in this article and, as you will see, defining new types would be easy (see Figure 1).

You can use the MailType property to identify the mail to be processed. When you create your mail, set the MailType property of your message to any value (its default value is to be unset).

When reading the mail, you can check the MailType property of each message to identify your "system managed" mail. Setting MailType also converts the mail to noninterpersonal mail so that it does not appear in your user's in-box.

I chose not to use MailType because I wanted to be able to create and read MailHandler messages on the fly from within the regular mail client. In case things went wrong, I also wanted to be able to audit the system just by reading the mail in our computers' in-boxes and sent-mail folders.

In addition, Microsoft's documentation suggests that MailType is not supported for all mail services providers. Finally, the receipt of non-IPM mail will not trigger the automatic execution of a program.

To deal with these situations, MailHandler checks for keywords in the subject line of mail messages using the InStr() function. These keywords determine if a message should be processed and what routine should be used to handle the message. InStr() allows the subject line of the message to contain other information (for audit purposes) and allows forwarded and reply mail to be processed by MailHandler.

Once MailHandler finds a message with a keyword heading, it opens the message and reads it, looking for more keywords. This means that when MailHandler wakes up and reads the mail, it can ignore mail that is already read because it's already been processed.

The ProcessMail routine creates a list of unread mail and runs through it looking for interesting subject headings. It processes the subject headings by calling an appropriate handler routine. When all the unread mail is checked, ProcessMail checks to see if any new mail has been received while it was busy and checks it. If none has, ProcessMail ends.

THE GOOD STUFF

Now that MailHandler can read the mail, what do you want it to do with the messages? The simplest action you probably want the program to take is to display a notification on the screen.

The DisplayNotification routine reads the text of the mail message looking for the keyword "Notify:" followed by a tab. Whatever follows the tab and precedes the next carriage return is displayed in a message box on the screen. A message with the Subject "Notify Message - a test message" with the text:

```
Notify:  Hello World
```

will cause MailHandler to display "Hello World" on whatever computer it is running on (see Figure 2). Here's the DisplayNotification routine:

```
Sub DisplayNotification(cntMail as _
   Control, ingPos as integer)
'passed a mail control, load the
'keywords array with the keyword to
'search for, get the value of the
'message and display it
Dim stzKeywords(1,2)
MAPIMessages.Index = ingPos
stzKeywords(1,1) = "Notify:"
ParseKeywords stzKeywords(), _
   (cntMail.Text)
If stzKeywords(1,2) > " " Then
   MsgBox Now() & ": " & _
      stzKeywords(1,2),16
End If
End Sub
```

Now you might ask, "What possible use could this routine be?" We have MailHandler running on a computer in our security guardhouse. Several jobs running on unattended computers at night send Notify-type mail to our guards when they reach check points (see Listing 2).

If the appropriate notification message doesn't appear on schedule, our guards call the appropriate people. A number of jobs also send mail when they abend or run into various other problems.

Our convention in MailHandler messages is to load the text with the repeating quartets of keywords, tabs, text, and carriage returns. The main reason for such

simple messages is that they were simple to create. Not only was a function to create these messages simple to write, but we can also create messages on an ad-hoc basis from within our normal mail client.

When reviewing either the in-box or sent-mail folders of any computer, it's easy to check sent and received messages, and whether each message was formatted correctly. Finally, we could write a simple ParseText routine that would accept an array of keywords and fill it with the corresponding values before returning it to the calling routine.

From within a program, creating a message consists of setting the Text property of your message. About the only thing you have to watch out for here are carriage returns (Chr(13)). If you're creating a message using the MAPI.VBX, don't *ever* begin the text of a message with a carriage return. VB will simply refuse to send your mail if you do.

RPC AND LICENSEE MANAGEMENT

A more interesting example consists of requesting another computer to run a program for you: sort of a postal Remote Procedure Call. One of our systems required our users to enter daily status information and then click on a button to process this information against a table of 2 million records. This processing tied up the user's computer for 10 minutes. We altered the program to send this message with the subject "Run Program—Do Update":

```
Program: P:\MASYSTEM\UPDATE.EXE
```

Recognizing that the subject contained the keywords "Run Program," MailHandler would call the RunProgram routine to process the message.

The simplest version of the RunProgram routine parses the keyword "Program:" and runs the program named after the tab. We enhanced the program by adding a routine to prevent MailHandler from moving to the next piece of mail until the program that was started had run to completion.

Not every process can use this facility due to the time lag between request and performance. Our mail clients can wait as long as 10 minutes to send mail to the server and, at the other end, our MailHandler computer can wait an additional 10 minutes before retrieving its mail. If either computer is busy, the delay can be even longer.

MailHandler won't respond to new mail until it has finished processing its current job. All these considerations mean that time-sensitive jobs can't be naively implemented using Mail. In this case, the job would be done in time if it finished before midnight, so it was a perfect candidate for MailHandler.

It's also important to recognize this store-and-forward process as a benefit. If the network is down, the mail client will hang on to the mail until it can be sent over the network. At the other end, if the receiving computer is down, the mail system will hang on to the mail until it can be sent. This fault tolerance ensures that the mail *will* get through.

Passing the command-line parameters to the program required a more complicated format. Our LAN-based applications often needed to transfer data to the AS/400, which was unable to handle ODBC access.

We wanted to limit the number of computers we installed the upload-to-AS/400 program on, so we installed it on a single computer along with MailHandler. Mail sent to this computer still uses the RunProgram message, but we enhanced the routine to handle a command-line keyword. The message:

```
Program: S:\SOFTWARE\TRANSFER.EXE
CommandLine:   P:\TRANSFER\BINFILE.TTO
```

VB3

```
Sub DisplayAttachment (cntMail As Control, ingPos _
  As Integer)
Dim stzAttachment As String, stzDefaultDir As String
Dim stzExecutable As String * 128
Dim ingResult As Integer
Dim obj As object
Dim ingEndPos As Integer
cntMail.MsgIndex = ingPos
stzAttachment = cntMail.AttachmentPathName
stzDefaultDir = "C:\"
ingResult = FindExecutable%(stzAttachment, _
  stzDefaultDir, stzExecutable)
If ingResult > 32 Then
  ingEndPos = InStr(stzExecutable, ".EXE")
  ingResult = Shell(Left$(stzExecutable, ingEndPos + _
    3) & " " & stzAttachment & Mid$(stzExecutable, _
    ingEndPos + 4, 20), 1)
Else
  MsgBox "Unable to find program to open " & _
    stzAttachment & "."
End If
End Sub
Sub SetExcelCommand (stzCommandLine as string, _
  stzDocument as string)
Dim obj As object
Set obj = CreateObject("Excel.Application.5")
obj.workbooks.open stzDocument
obj.Range("Command").value = stzCommandLine
obj.activeworkbook.[Close] True
obj.quit
Set obj = Nothing
End Sub
Sub SetWordCommand (stzCommandLine as string, _
  stzDocument as string)
Dim obj As object
Set obj = CreateObject("word.basic")
obj.fileopen stzDocument
obj.setdocumentvar "Command", stzCommandLine
obj.fileclose 1
Set obj = Nothing
End Sub
```

LISTING 3 ***All Set!*** *SetExcel and SetWord provide a standard way to pass data to Word and Excel. It's important to set the object variables to Nothing before ending the routines. If you don't do this, both applications are prone not to give up their resources or, in Excel's case, not shut down at all.*

A Brief Introduction to the MAPI Family

The MAPI family consists of three siblings: Simple MAPI, Extended MAPI, and Common Messaging Calls (CMC). The MAPI.VBX rests on Simple MAPI, which has unfortunately become the family's black sheep. CMC is the younger, more favored half-sister, while Extended MAPI is the more capable big brother.

MAPI is designed to allow Windows to do for electronic mail what it has already done for printing—remove device dependence. If you use MAPI calls in your program to implement electronic mail, your application should be able to work with any combination of vendors' address books, message stores, and transport services. Like printing, MAPI is a complete subsystem within Windows with a set of front-end interfaces for your application to use and back-end interfaces for e-mail services. In the same way that the printing subsystem makes plotters and printers look alike to your application, MAPI makes fax services and bulletin boards appear the same. Like the printing subsystem, the MAPI subsystem has its own spooler. MAPI also breaks e-mail down into four separate kinds of services—address books, message stores, transport services, and profile services (which allow users to specify which combination of services they require). MAPI allows application developers to work with any combination of MAPI-compliant service providers.

MAPI was created by Microsoft in consultation with industry vendors as part of the Windows operating system and services. CMC was developed later, in conjunction with the X.400 API Association. Because it is based on X.400 specification, it's a platform- and OS-independent specification. Microsoft's recommendation is that you use Simple MAPI only if you have legacy Simple MAPI applications to support.

CMC is designed to allow developers access to basic mail functions from a wide set of tools—from application macro languages to C++. Unlike ODBC, MAPI has no conformance levels other than the distinction between the minimum level represented by the 10 CMC calls and the full functionality of Extended MAPI. It is the developer's responsibility when working with Extended MAPI to determine what functionality is available from the e-mail services.

From within Visual Basic, MAPI VBX represents the easiest access to the functions of CMC. Similarly, within Access and Excel it's easy to use the SendObject action and SendMail method, respectively. The Word Resource Kit provides functions for using MAPI within Word, or you can use the workarounds suggested in the article. Microsoft may choose to implement the functionality of these high-level objects through any of the MAPI family without impact on your applications. However, in other environments you may have to choose between CMC and Extended MAPI calls.

Microsoft distinguishes between three kinds of mail-enabled applications. Mail-aware applications are those that make mail operations available as part of their feature set. The ability to route a Word document through several users is a typical example. These applications will probably find all their needs met with CMC. Mail-reliant applications need mail to perform their functions, while workgroup applications are, in many ways, high-level mail processors. These applications should probably look to Extended MAPI. The distinction often comes down to whether the application needs to read mail. If it does, more often than not it is going to require Extended MAPI.

In addition to being object oriented, Extended MAPI provides many features that CMC does not. Three of the more important are:

- **Access to the hierarchy of folders. CMC allows the developer to view only the contents of the user's in-box. The developer cannot view, create, or delete other folders, or move messages between folders.**
- **Search capabilities. The search capabilities within Extended MAPI allow the developer to build complicated cross-folder searches and display the results of the searches in a new "search folder."**
- **Event notification. Extended MAPI lets applications request that MAPI notify them when certain events occur, eliminating the need for polling.**

A similar initiative is the VIM (Vendor Independent Mail) specification developed by Borland, Lotus, Apple, and IBM. Many mail-service providers support both specifications.—P.V.

launches the transfer program passing the name of the transfer specification file (BINFILE.TTO) on the command line. Needless to say, the first version of this new routine worked well when a command line was specified, but crashed when it got one of the old messages with no command-line parameter.

ROAD WARRIOR SUPPORT

While the RunProgram routine would handle a great many situations, we had to add a new routine to handle Microsoft Access. Perhaps your sales staff needs to request from the database information regarding sales to a customer, typically before visiting that customer. While we wrote an Access application to provide the information, it runs too slowly over dial-up links to be useful to salespeople on the road. A program to create and mail this message with the subject "Run Program—Get Sales Status" takes about 20 minutes to write in VB, and less time in Access:

```
Program: MSACCESS.EXE
Document:   P:\MASYSTEM\ORDERS.MDB
CommandLine:   145678 Wbrimley
Macro:      GetOrderInfo
```

Here, the command-line data consists of the customer number and the e-mail address of the salesperson requesting the information. When a computer at corporate headquarters running MailHandler receives this mail, the RunProgram routine is called. RunProgram parses the message, starts Microsoft Access with the specified database, and uses Access's /X command-line parameter to specify the macro to run. It will also use the /C parameter to pass data to Access on the command line.

The GetOrderInfo macro that the message requested to run is three lines long and consists of the actions:

- SetWarnings, to turn off Access's warning messages.
- SendObject, to create and mail the report as an Excel spreadsheet.
- Quit, to exit Access after the job is run.

The query that generates the report must be altered to parse the first word of the command line and use it as the criteria to select the customer:

```
Left$(Command$(),Instr(Command$()," ")-1)
```

The SendObject action must parse the second word in the command line to get the address of the salesperson to mail the report to:

```
Mid$(Command$(),Instr(Command$()," ")+1)
```

Your road warriors send their requests and get their answers by return e-mail.

If several pieces of mail are to be sent and processed in order, be careful. While MAPI can sort the mail in order by time sent, the precision of the time can't be any finer than to the minute. As a result, if two messages are sent in the same minute, their processing order is not guaranteed. When we needed to send more than one piece of mail and needed the messages to be processed in a specific order, we had to ensure that they were sent at least a minute apart.

In Word, you specify the startup macro with the /M switch. In Excel, you can specify only one routine to be run when a workbook is opened. For workbooks that have several different routines, an Auto_Start routine must parse the command-line information and call the appropriate routine.

Neither Word nor Excel support the Command() function from Access and Visual Basic, so data cannot be passed through command-line parameters. The simplest workaround is to use OLE Automation in the RunProgram routine of MailHandler to

set a cell called Command in the Excel worksheet to the passed values. In Word, setting a document variable accomplishes the same task (see Listing 3).

The RunProgram routine checks Access, Excel, and Word as the specified program to format their command lines correctly and set the passed data. It might have been better practice to create separate routines with their own keywords and message formats.

Creating return mail in both Excel and Access is simple. The SendMail method in Excel and the SendMail Action in Access are the single commands required to mail workbooks, forms, reports, or tables.

In Word, however, sending mail is more difficult. One solution is to use the Word Resource Kit, which includes a Word DLL with the necessary MAPI routines. The Word Resource Kit is available in *The Word Developer Kit*, Second Edition, Version 6.0 from Microsoft Press (ISBN 1-55615-681-2) and in the Microsoft Office Developer's Kit. As a workaround, you can create a Word macro using Word's MailMerge commands.

While the required Word macro is only a few lines long, before you can use it you must create a Word data document containing the Address1 field. The Word MailMerge Wizard lets you do this: create a data document called MAILDUM.DOC. Put a copy on each computer running MailHandler. At mail time, the macro opens MAILDUM.DOC and changes the Address1 field to the e-mail address of the person to be mailed to. It then closes MAILDUM and mails the document:

```
Sub MailIt(address$, subjecttext$)
FileOpen  .Name = _
   "c:\msoffice\winword\maildum.doc"
NextCell
Insert address$
FileClose 1
MailMergeOpenDataSource
    .Name = "MailDum.doc"
MailMerge .Destination = 2,
   .MailSubject = subjecttext$,
   .MailMerge,
   .MailAddress = "Address1"
End Sub
```

MailSubject can be set to any string you like and will be used as the subject line of your mail message. Using a menu selection from Word's General Options menu, you can choose to have the document sent as the body of your message or as an attachment.

THE USER INTERFACE

Having gone this far, the next step was to move out of merely batch processing and into an interactive world. We were already mailing various Access reports to our users as Excel spreadsheets or attaching Word documents to our mail.

However, our users were having difficulty finding and working with this output when their in-boxes got full. Putting a front end on MailHandler turned out to be relatively easy and allowed our users to find and process their computer-generated mail more easily.

The first step was to enhance the Form_Load procedure to accept another parameter to display the user interface:

```
Sub Form_Load
If command() = "/I"
   frmUI.Show
ElseIf command() = "/T" Then
' etc.
End Sub
```

The second step was to add a line to the MSMAIL.INI file to put "Process Mail" on the Tools menu of the user's Mail client. When selected, this choice calls MailHandler with the /I parameter:


```
PWF=3.0;Mail;&Process Mail...;13;_
   APPEXEC.DLL;C:\MAILHAND.EXE /I;;_
   Process workflow mail;_
   MSMAIL.HLP;0000
```

If the user needs to interactively process received mail, you should remove the line in the MAIL.INI file that causes mail to process automatically (see Figure 3).

With this housekeeping out of the way, we began development on the workflow front end to MailHandler. Previously, the system retrieved all unread mail and processed it immediately. The workflow screen would have to retrieve all the mail and process it on request. To handle this, the workflow screen retrieved the unread mail with interesting keywords and loaded the subject line to a standard grid control.

Along with the subject information, the position of the message in the mail queue was stored in the grid. A double click on any cell on the grid calls the standard ProcessMail routine, passing the MAILMESSAGE control and the position of the selected message in the queue as usual (see Figure 4).

Members of the systems group in my organization were the first users of the screen. When mail needed to be reprocessed, or processed out of order, the workflow screen made the job much easier.

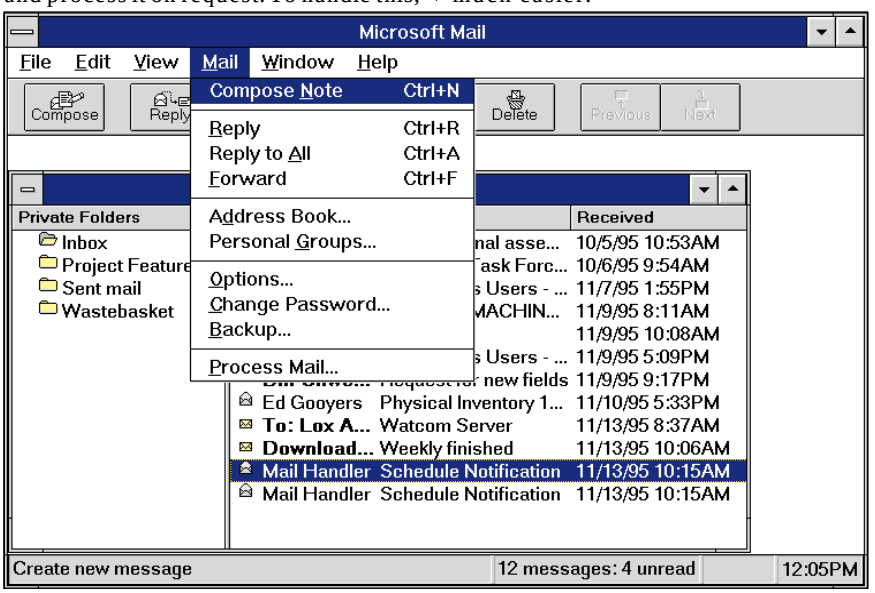

FIGURE 3 ***The Process for Processing Mail.*** *A single line in the user's MSMAIL.INI file allows the user to start an interactive session with MailHandler. Clicking on "Process Mail" starts MailHandler with the /I option, causing it to display a list of all the mail it finds interesting.*

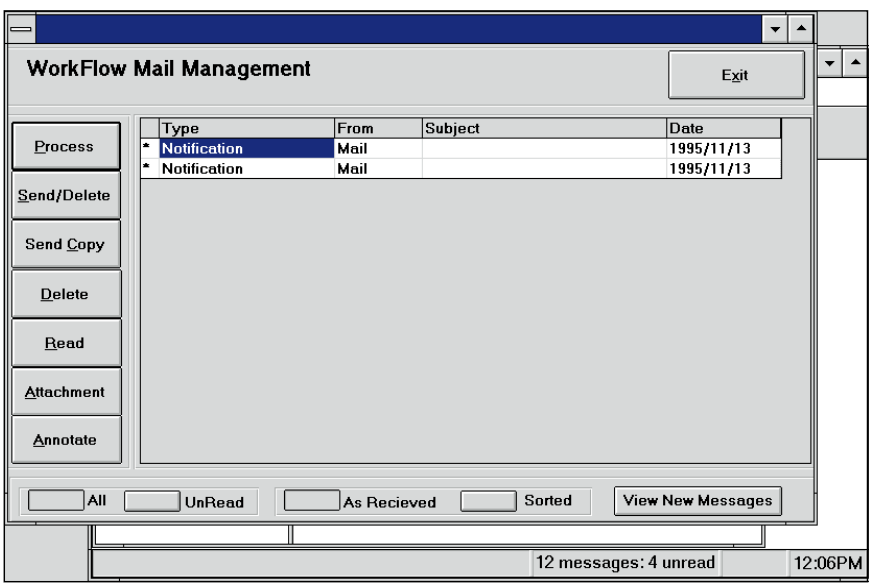

FIGURE 4 ***Junk Mail?*** *The workflow screen allows the user to review his or her computer-generated mail and process it. In addition to processing the message, the user can delete the mail, read it, forward it, and add annotations.*

The next application was for our order-processing system. When an order requires financial approval, a mail message would be sent from the order-entry application to our finance manager. Once received, the mail message opens the application displaying the sales order whose order number was passed on the command line:

```
Program: Ordentry.Exe
CommandLine:   17406
```

We added a "Display Attachment" keyword to the system to allow our users to review e-mailed reports without having to search them out in their mailboxes, fulfilling the original objective in adding the interactive processing.

Now that you've started sending mail to particular users, it's important that you do *not* use their e-mail addresses as the "To:" line of the message. If you do, each time someone changes jobs you will have to rewrite your code.

Instead, set up recipient lists named for the job that is to be done and direct the mail to the list name. While most lists will have only one member, this practice allows you to have mail sent to multiple users where necessary. As people change jobs and jobs are reassigned, you can use your mail system's administration facility to change the person who belongs to the job.

At the very least this saves you from having to develop the facility yourself. The only documentation that must be maintained is a list of which job streams are associated with which recipient lists.

This is only a small peek at what mail can do for you. We are starting to keep our documentation and company guidelines in Windows help files. At the top of each help screen is a button that allows the user to mail comments, corrections, and additions to the help file's manager.

A database of reviewers and interested parties for each help file helps keep track of who should be notified (by mail, of course) when the file is updated. The interactive screen allows the responsible person to manage and track these requests.

I keep a schedule of what jobs are to have been run at what times over the course of the day. MailHandler writes to a log file every time it performs an activity.

Periodically the system runs a job that checks to see if some scheduled job hasn't run on time and mails a "Notify" message to the responsible person (note: many mail systems will automatically forward the mail to a delegate if a person is away). Also, we can run a monthly report comparing MailHandler recorded activity against the schedule. For a lot of problems, the answer is in the mail. ×