FT-V02

Outlook Automation for Visual FoxPro

Andrew MacNeill
AKSEL

Overview

When we first learned about Automation in FoxPro 2.x and Visual FoxPro, it was usually using Word and Excel. But those are simply two examples. No application is an island anymore. By accessing other COM servers, your application can become infinitely more powerful. By having other applications access YOUR COM server, you can concentrate on business logic instead of interface design.

This session uses Microsoft Outlook as both a primary and secondary COM server and discusses the different ways that Outlook and Visual FoxPro can work together to accomplish a myriad of functions.

The target audience for this session is the Visual FoxPro developer who:

- Has done some basic automation, perhaps using Word or Excel
- Is familiar with the syntax of Visual Basic or VBA code
- Has heard about or built their own COM server
- Is tired of rebuilding the wheel for every application

Code re-use is one of the reasons many developers like Visual FoxPro. The object-oriented features serve developers well, reducing the code required to perform various functions. Re-usability isn't limited to code. Many times when building an application, I pull forms and ideas from other applications, reusing these pieces as well. Instead of just sticking to existing code, why not consider using an existing application to reduce all your hard effort?

When deciding to include automation in your app, you have to consider how your users will be using it. If your application is already established with your users and has a known user interface, it's probably best to link to another application to provide additional functionality. In this scenario, you control the interface and you have complete control over what the user sees.

On the other hand, if you are building a new application that mirrors functionality in an existing office application, consider manipulating that application to call yours. Training, which should always be a consideration when building new applications, can be greatly minimized. The amount of work that you have to spend on the user interface will also be less, freeing you to concentrate on the business logic.

As an end-user application, Outlook is used to:

Send emails

Manage contacts

Schedule Appointments

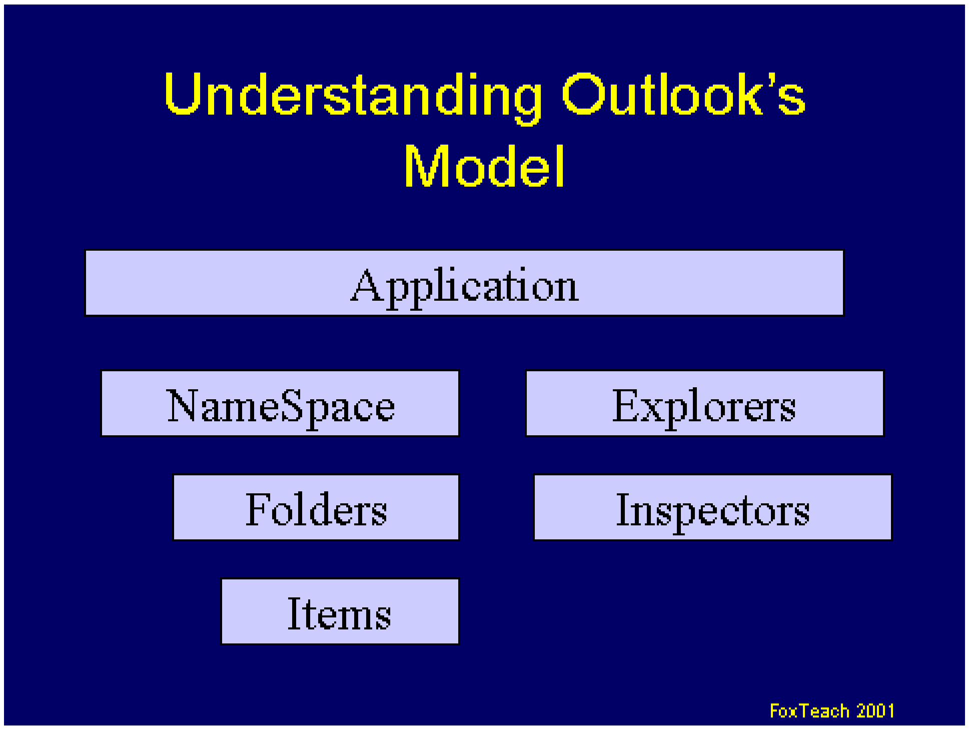
Manage Tasks

And more

Think about the number of applications you may have built that had to do a number of these sames tasks! An application that deals with tracking people probably could benefit with those people being in Outlook. Outlook already has sort and grouping functionality, query capabilities, and better yet, it merges with Word easily. An application that tracks resources or provides scheduling could use Outlook to display calendars, print filled in schedules as well as Outlook's conflict resolution features. It could also use Outlook's Task features to handle basic task management. All of these things are already handled in Outlook and with a little work can be handled easily within your application as well.

This session shows you how.

The Outlook Object Model



The Outlook object model starts at the Application level. The Outlook Application object exposes a few properties that are used to access the rest of Outlook, the most important one is the NameSpace object.

The NameSpace object is a reference to the current Outlook data source or session. It only has one supported data source which is MAPI. From the namespace object, you can access all of the Folders and items within Outlook as well as AddressLists.

Items reside within folders. Outlook items include the following:

- Appointments
- Contacts
- DistList (Distribution List)
- Document
- Journal
- Mail Messages
- Meetings

- Notes
- Post
- Remote
- Report
- Task (including requests)

Each item has its own properties and key methods

To create a link to Outlook, the following steps are required:

- Create the Application Object.
- Get a pointer to the current session.
- Find the folder that you want to work with.
- Go to work.

The code to do this is also relatively straight forward:

```
loApp = CREATEOBJECT("Outlook.application")
loSpace = loApp.GetNameSpace("MAPI")
loFolders = loSpace.Folders
loItems = loSpace.Folders.Items
```

The loApp statement creates the Application object. LoSpace returns a reference to the current Outlook session.

Moving through folders with Outlook can be done using the GetDefaultFolder method or by traversing the Folders collection.

As with most collections, you can specify the name of the folder directly or specify it by index count:

```
FOR lni = 1 TO loSpace.Folders.Count
   lcName = loSpace.Folders(lni).Name
ENDFOR
Folders are hierarchical so each folder can contain their own set of
folders as well.
FOR lni = 1 TO loSpace.Folders.Count
   loFolders = loSpace.Folders(lni).Folders
   FOR lnSub = 1 TO loFolders.Count
      lcName = loFolders.Folders(lni).Name
   ENDFOR
ENDFOR
```

In most Outlook sessions, however, it is much easier to use the GetDefaultFolder method which returns the user's commonly used folder. This setting is controlled by where the user retrieves messages from.

The GetDefaultFolder method returns the default folder based on a passed parameter:

3 – Deleted Items

4 – Outbox

5 – Sent Items

6 – Inbox

9 – Calendar

10 – Contacts

11 – Journal

12 – Notes

13 - Tasks

```
loInbox = loSpace.GetDefaultFolder(6)
```
Every Folder has an Items collection. Items.Count returns the number of items. The Add method creates a reference to a new item. The item however is not saved until you do something with it.

```
loItems = loFolders.Items
loNewItem = loItems.Add( )
```
To immediately create user interaction, you can display the item right after you create it.

```
loNewItem.Display ( )
```
However, when an item is displayed, the user now has control over it, meaning they can destroy, delete or send it without giving your application the time to react.

When you have finished working with an item, you can either Save or Send it. Sending is an action reserved for those items that have a communication context. For example, you don’t Send a contact, you save a contact. Likewise, you usually wouldn’t save a message; you would send it.

Common Item Methods

All items have standard methods that can be used for moving items around.

The Delete method deletes an item. Deleted items are placed in the Deleted Items folder. To permanently delete items, delete items that are in the Deleted Items folder.

```
TrashFolder = loSpace.GetDefaultFolder(3)
lnItems = trashfolder.items.count
For lni = 1 TO lnItems
   Trashfolder.items(0).delete()
Endfor
```
Call the Copy method to create a copy of the message.

```
myCopy = loItem.Copy()
```
The Move method then moves the message into another folder.

```
SaveFolder = loSpace.Folders("My Mail").Folders("Important Messages")
MyNewMsg = myMsg.Copy
MyNewMsg.Move SaveFolder
```
Note that the SaveFolder isn't a string but a pointer to the desired folder.

Attachments

Attachments are listed in an item's Attachments collection. You can add an attachment as a shortcut to the item, a direct attachment or as an embedded Outlook object.

```
myNewMsg = myFolder.Items.Add()
myNewMsg.Attachments.Add("C:\MYAPP.ZIP",1,1,"Compressed Application Zip
File")
myNewMsg.Attachments.Add("\\aksel1\akseld\Info.doc",2,1,"Link to
Information document")
```
The first Attachment is the actual file. The second attachment is a shortcut to the file.

The parameters to the Add method are displayed below.

Parameter Name	Description
Source File	Full path name of attachment, URL or object being attached.
Type	Type of attachment: 1 – Separate file Attachment 4 – Link or shortcut to attachment

	5 – Embedded object
Position	Position in body where attachment goes. A value of 0 does not send the document.
DisplayName	Name that will be displayed to the user

The add method only requires one parameter but using all 4 provides more functionality.

When receiving a message with attachments, call the SaveAsFile method to copy the attachment to another location. To remove the attachment after copying, call the Delete method.

```
FOR EACH loAttachment IN loMsg.Attachments
   IF loAttachment.Type = 1
   loAttachment.SaveAsFile("Downloads\"+loAttachment.FileName)
   ENDIF
ENDFOR
```

The FileName property returns the short filename. The PathName returns the full path to the attachment. The DisplayName is the name that was displayed in the message.

The Message Item

The table below displays most of the commonly used properties for a message item.

When you create a message, the most likely method to use is Send. However, if you call Save, it will save a copy of the message.

Property	**Description**
Subject	Name of the task
SenderName	Name of sender
Body	Content of message
Attachments	Attachments collection (see below)
Recipients	Recipients collection (see below)
Importance	Importance Flag 0 – Low 1 – Normal 2 – High
Sent	Logical value. Identifies if message was sent (always True for received messages)
SentOn	Date and time message was sent
ReceivedTime	Date and time message was received
CC / BCC	Name of CC and BCC names. The actual individuals come from the Recipients collection.
Companies	Name of company associated with item (free-form).
ConversationIndex ConversationTopic	The topic and index of the conversation (see below).
CreationTime	Date and time message was created
EntryID	Unique identifier for message
ExpiryTime	Date and time message can be declared expired and can be deleted.
FlagStatus	Flag Status for message. 0 – No Flag 1 – Completed 2 – Flag Marked

FlagDueBy	Date and time flag is due. FlagStatus must be 1 or 2 for this property to be valid.
FlagRequest	Requested action for message. Free-form text
HTMLBody	HTML version of the body. Changing this will change the body property.
ReplyRecipients	Collection of recipients who will receive the message if you reply to it.
Sensitivity	Sensitivity of message. 0 – Normal 1 – Personal 2 – Private 3 – Confidential
Unread	Logical value indicating if message has been read or not.
VotingOptions VotingResponse	Voting options (see below)

The ConversationTopic and Index properties can be used to group similar messages together. When a message is replied or forwarded, the subject is updated with the words "RE:" or "FW:". The ConversationTopic property however maintains the original topic. So in the case of a series of messages as shown in figure 1, the conversation topic is always the same. The ConversationIndex property is a string. It is supposed to contain the Index of the conversation. However, in Visual FoxPro, the string is made up of unreadable characters. New messages return an empty ConversationIndex.

Why is the ConversationTopic property important? A message may not have the same Subject and yet belongs to the same conversation. Outlook does not make it easy for a user to retrieve messages by conversation, however, Visual FoxPro can!

```
LOCAL ln
ln = 1
DIMENSION laConversations(ln)
LaConversations(ln) = ""
FOR EACH loMsg IN myFolder.Items
   IF ASCAN(laConversations,loMsg.ConversationTopic)=0
      DIMENSION laConversations(ln)
      laConversations(ln) = loMsg.ConversationTopic
      ln = ln+1
   ENDIF
ENDFOR
MESSAGEBOX("You are having "+LTRIM(STR(ALEN(laConversations,1)))+"
conversations in this folder.")
```

Replying and Forwarding Messages

Call the Reply method to create a message that has all of the recipient information filled in based on the current message.

Call the Forward method to create a message that contains all of the content of the current message.

When replying or forwarding, the original message is/is not included in the body. Therefore, be careful of the changes made to the Body property. They may affect the original message.

```
newMsg = loMsg.Reply
newMsg.Body = "I agree 100%." && This overwrites the original message
newMsg.Send()
newMsg = loMsg.Reply
```

```
   newMsg.Body = "I agree 100%."+ CHR(13) + newmsg.body
   newMsg.Send()
```
The second example maintains the body of the original message.

Recipients

When sending and receiving messages, the Recipients collection contains any person receiving the message, including any CC (carbon copys) or BCCs. Change the Type property of the recipient to set a recipient as a particular type. The default type is 1 (which is the To column). CCs and BCCs are types 2 and 3 respectively.

Once a recipient has been added, call the Resolve method to verify the person. The Resolve method checks the address book for the recipient, ensuring that the message can be sent.

```
   newMsg = myFolder.Items.Add()
   newMsg.Subject = "Welcome!"
   oSendTo = newMsg.Recipients.Add("Andrew MacNeill")
   oSendTo.Type = 1  && To
   IF NOT oSendTo.Resolve()
      MESSAGEBOX("Recipient is invalid")
   ENDIF
```
If you are using Internet email addresses, the Resolve method can be bypassed, as it will be used automatically when sending the message.

```
   oSendTo = newMsg.Recipients.Add("testmessage@aksel.com")
   oSendTo.Type = 2
   newMsg.Send()
```
When reading messages, the Recipients collection shows all of the people who received the message. Each Recipient in the collection has thee properties: Name, Address and Type. Other Recipient properties are listed below.

Property	Description
Name	Displayed name of the recipient
Address	Email address of the recipient
AddressEntry	Pointer to the AddressEntry object in the AddressList collection (see below).
DisplayType	Nature of the recipient: 0 – User 1 – Distribution List 2 – Forum 3 – Agent 4 – Organization 5 – Private Distribution List 6 – Remote User
Resolved	Logical value indicating if name was resolved or not.
TrackingStatus	Tracking status of the message. 0 – No Tracking

	1 – Track Delivered 2 – Track Not Delivered 3 – Track Not Read 4 – Track Recall Failure 5 – Track Recall Success 6 – Track Read 7 – Tracking Replied If set to other than 0, tracking messages will be sent to the sender as different information is received.
Type	Type of recipient. 0 – Originator 1 – To 2 – Carbon Copy 3 – Blind Carbon Copy

Contacts

The Contact Item represents a placeholder for any information you want to track about a person. For a single contact, you can store 3 mailing addresses, 3 e-mail addresses, 19 different contact numbers and many more pieces of personal information. Most of the fields are well-named but they tend to get a little long (see the next table for a list of commonly used contact fields).

Property	**Description**
FirstName	First name of the contact.
LastName	Last name of the contact.
FullName	Automatically combines firstname and lastname properties.
CompanyName	Company name.
MailingAddress BusinessAddress HomeAddress OtherAddress	The address including street, city, state, zip code and country, if entered.
xxxxAddressStreet xxxxAddressCity xxxxAddressState xxxxAddressPostalCode xxxxAddressCountry	Any component of the address where xxxx is Mailing, Business, Home or Other.
BusinessFax BusinessTelephoneNumber Business2TelephoneNumber HomeTelephoneNumber Home2TelephoneNumber MobileTelephoneNumber	Various telephone number fields. Note the lengths of the properties are fairly long which makes it very easy to misspell.
Title	Fields used to identify position or company

Department Account	information.
Birthday	Birth date in Date/Time format.
Body	Unlimited text similar to a memo field.

Appointments and Calendars

The Appointment Item is the Outlook item that appears in the Calendar views.

Property	**Description**
Subject	The title of the meeting
Body	Details of the appointment
AllDayEvent	Logical field indicating if event is an all-day event.
BusyStatus	Numeric field indicating how the appointment appears. Values are: 0 – Free 1 – Tentative 2 – Busy 3 – Out of Office
Duration	The duration (in minutes) of the appointment.
Start	The start date and time of an appointment.
End	The end date and time of an appointment.
Importance	A numeric value indicating the importance of an appointment. Values are: 0 – Low 1 – Normal 2 – High
IsOnlineMeeting	Logical field indicating if meeting is online. This doesn't mean anything unless you decide to use it.
IsRecurring	Logical field indicating if item is a recurring appointment (see below).
Location	The location of the meeting.
MeetingStatus	Numeric value indicating the status of the appointment. By setting the MeetingStatus, you can make an appointment show on a network calendar. Values are: 0 – Non Meeting 1 – Meeting 2 – Meeting Cancelled 3 – Meeting Received
Organizer	Returns the name of the Organizer of the appointment.

Scheduling Meetings

To schedule a meeting, you must **send** the Appointment item to the appropriate recipients as well as Save it.

Use the Recipients collection to schedule a meeting with participants. The Type property identifies whether an attendee is required (1), optional (2), the organizer (0) or a resource (3).

```
    myAppt = oCal.Items.Add()
    myAppt.Subject = "DevCon Party"
    myAppt.Start = CTOT("11/25/2000 17:00")
    myAppt.Duration = 120   && two hours
    oPerson = myAppt.Recipients.Add("Randy Brown")
    oPerson.Type = 1
    oPerson = myAppt.Recipients.Add("Steve Black")
    oPerson.Type = 1
    oPerson.Resolve()
    oPerson = myAppt.Recipients.Add("Calvin Hsia")
    oPerson.Type = 2
    oPerson.Resolve()
    oRoom = myAppt.Recipients.Add("DevCon Attendees")
    oRoom.Type = 3
    oRoom.Resolve()
    myAppt.MeetingStatus = 1
    myAppt.Send()
    myAppt.Save()
```

When you add recipients, call the Resolve method to ensure that the names are found properly in the Outlook Address book. If the Resolve method isn't called, the meeting request may be delayed in sending.

In the above example, the last Recipient isn't a single person: it's a Distribution list. As long as these names can be resolved within the current AddressList, the recipient is considered valid. If a recipient isn't valid, then the message will not be sent.

Set the MeetingStatus property to 1 before sending the meeting request. The default property value is 0 which means that requests will not be sent to attendees. To cancel the meeting, set the property to 2 and send the message again.

Once the meeting has been sent, refer to the ResponseStatus property to see how the attendees have responded.

Call the Remove method in the Recipients collection passing it the index number of the recipient to remove from the list.

```
    myAppt.Recipients.Remove(3)
```

After identifying the recipients, use the Appointment properties identified in the table below to differentiate between them.

OptionalAttendees	Strings that display the Display Name of the attendees. Although you can modify these properties, use the Recipients collection to add and remove participants.
RequiredAttendees	
Resources	
ResponseStatus	The response to the meeting requests. Once one person responds, this property is set. 0 – No response received 1 – Organized 2 – Tentative 3 – Accepted 4 – Declined 5 – Not Responded

Tasks

In Outlook, a Task is a message that provides details about something that needs to be done. If you work on a project, there might be several tasks that need to be done. If you work alone, there might be several projects (all containing several tasks) that you are working on.

Each task has its own set of properties that can be used to manage it. Outlook can even be used to manage a mini-project. Even if you already use Microsoft Outlook to manage your tasks, you might not be aware of all of the properties (most of them don't show up when looking at a task). The table below lists the various properties, starting with the standard properties.

Property	**Description**
Subject	Name of the task
StartDate	Date the task was started
DueDate	Date task is due
Status	Status of the task
Priority	High/Medium/Low
PercentComplete	% of the task that has been completed
Body	Notes for the task
ReminderSet	Logical field indicating if task has a reminder set for it.
ReminderTime	The date and time that the reminder will be played.
ReminderMinutesBeforeStart	The number of minutes the reminder should occur before the start of the appointment
ReminderOverrideDefaults	Logical field indicating if task has its own settings (instead of using the default Outlook settings).
ReminderPlaySound ReminderSoundFile	PlaySound is a logical field indicating if the reminder should play a sound. SoundFile is the full path name of the WAV file to be played.
DateCompleted	Date the task was completed,
TotalWork ActualWork	A numeric field representing the number of minutes the task took. Outlook converts the time automatically to minutes, hours or days depending on the duration.
Mileage	String or numeric used to represent mileage required to perform task.
BillingInformation	Text field for additional billing-related information.
Companies	Text field for additional company related information.
Role	Text field for the role the user is playing in the task.
TeamTask	Logical field indicating if Task is being performed by a team.
Owner	Indicates who is the owner of the task.
Ownership	Read-only numeric value reflecting the state of the task. 0 – New Task 1 – Delegated Task 2 – Own Task
Categories	Comma-delimited list of categories assigned to task.
Complete	Logical field indicating if task is complete. If True, PercentCompleted is automatically 100%.
DelegationState	Read-only numeric value reflecting how the task was delegated.

	0 – Task Not Delegated 1 – Delegation Status Unknown 2 – Accepted 3 – Declined
Delegator	Read-only string returning the name of who delegated the task.
Recipients	Collection of other Outlook users who will receive the assigned task (see Recipients).

Delegating Tasks

You can also delegate tasks using Outlook. When you delegate a task, you are, in effect, sending them a message with the details of the task, asking them to take ownership of it.

The following code delegates a task to another user:

```
loTask = oTasks.Item(1)
loTask.Recipients.Add("John Smith")
loTask.Assign()
loTask.Send()
```

When a task has been delegated, the DelegationState property is immediately updated to 1. This value indicates that Outlook is unaware whether or not the person has actually accepted the delegation of this task. If the assignee (in this case, John Smith) refuses the assignment, the DelegationState is automatically updated to 3. If accepted, it is set to 2.

The following code returns a list of tasks, waiting to be assigned:

```
FOR EACH loTask IN oTasks.Items
   IF loTask.DelegationState = 1
      ? loTask.Subject
   ENDIF
ENDFOR
```

If you plan on assigning a task to a person and then decide against it, call the CancelResponseState method to return the task back to a simple task.

```
loTask.Assign()
loTask.CancelResponseState()
loTask.Save()
```

Using Defined Properties

If the fields provided still don't meet your requirements, you can create user-defined fields. User-defined fields are stored with each item individually. One contact may have the field where others may not.

This code creates a custom field named BalanceDue for a contact.

```
loContact = loCustomers.Item(1)
loContact.UserProperties.Add("BalanceDue",1)
```

UserProperties is a collection within each item. It has a Count property so you can see how many custom fields are attached to an item. What's particularly useful about the UserProperties collection is that if you attempt to add a field that already exists, Outlook doesn't error out.

The first parameter to the Add method is the name of the custom field. The second parameter is the type of field you are adding. Table 3 shows a list of different field types.

Type	**Description**

1	Free-form text
3	Number
5	Date/Time
6	Logical value.
7	Duration (defaults to minutes) entered as a number.
11	Keywords
12	Percent
14	Currency
18	Formula
19	Combination (similar to text).

You can specify that a custom field must be one of the types above.

Set the Value property to update a custom field.

```
   loContact.UserProperties("BalanceDue").Value = 500.50
```

Call the Delete method to delete a user property.

```
   loContact.UserProperties("BalanceDue").Delete()
```

Finding Information

The Outlook object model makes it easy to navigate through the contacts folder. The GetFirst and GetLast methods return references to the first and last items in the folder respectively. After moving to the first or last item, call GetNext or GetPrevious to move up and down through the folder.

Call the Find method to locate a contact. Pass the criteria as the only parameter.

```
   loFound = loCustomers.Items.Find("[LastName]='MacNeill'")
```

The syntax for writing filters is property name, surrounded with square brackets and the search term. You can create more complex search criteria with the keywords AND, OR and NOT.

```
   loFound = loCustomers.Items.Find("[MailingAddressState]='CA' AND
   [BalanceDue]>500")
```

You can combine user-defined properties and regular properties in the search condition. Outlook will parse the search string and find the correct result. If there were no items matching the search criteria, the Find method returns NULL. Call the FindNext method to find the next occurrence based on your current item position.

```
   loFound = loCustomers.Items.Find("[BusinessAddressState]='CA'")
   DO WHILE NOT ISNULL(loFound)
      ? loFound.FullName
      loCustomers.Items.FindNext()
   ENDDO
```

Displaying Information

Updating information in Outlook invisibly is useful in Outlook however, you can also access methods to display information. Displaying is done with the help of the Explorer and Inspector collections. An Explorer, like its Windows counterpart, is used to display lists of information. For example, a list of messages or contacts is displayed in Outlook with an Explorer. When you double-click a contact, it appears in a form that is controlled by an Inspector.

The Application object has ActiveInspector and ActiveExplorer properties to help identify what view the user is currently looking at. However, Outlook is generally a modeless application and users can have multiple windows open at once. You can identify the inspector or explorers with GetInspector and GetExplorer methods as well.

Inspector and Explorer objects have some common properties and methods:

Caption – returns the form caption displayed to the user

Close – closes the form

Display – displays the form or refreshes the view

Activate – activates the form

The Explorer object also has CurrentView and CurrentFolder properties. CurrentFolder returns an object pointer to the current folder being viewed. CurrentView is the name of the view being displayed. For example, Inbox messages are usually viewed with the “Messages” or “Unread Messages” view. Contacts may be viewed with “Contact Cards” or “Phone List” views.

The Inspector object has a CurrentItem property which returns the current item being looked at.

Accessing VFP from Outlook

So far, I have concentrated on automating Outlook from within our existing application. That approach assumes that you already have an application built, users are comfortable with it and you are basically "adding" functionality".

Consider the new project. Building Outlook-like features with their own interfaces is a massive project on its own. If you can make use of Outlooks' features and then add functionality to them, you can build a powerful application with an interface users already know.

In addition, training can become a breeze as you are simply extending their knowledge of an existing application.

Experienced User: Well, you need to learn our Employment Management resource system.
New User: Should I schedule training?
EU : Do you know MS Outlook?
NU : Yes.
EU: End of discussion.

The first step in accessing VFP from Outlook is to build a COM server that will be accessed from Outlook.

For this example, we are going to build a simple COM server application and build it up from there. Our COM server is going to act like an Audit Trail for certain types of activities in Outlook.

In addition, we are going to build some special forms for our users that include the information we need to track.

Here is the basic code for the first version of our COM Server. It has a single public method named WriteLog that uses the VFP function STRTOFILE to place information into a log file.

```
DEFINE CLASS OutTrack AS custom OLEPUBLIC
Name = "OutTrack"
PROCEDURE writelog
   LPARAMETERS tcInfo
     IF EMPTY(tcInfo)
       tcInfo = ""
     ENDIF
     =STRTOFILE(tcInfo+" "+TTOC(DATETIME())+;
       "   "+SYS(0)+CHR(13),"\OUTLOOK.LOG",.T.)
ENDPROC
ENDDEFINE
```

Create a project named OUTCOME and add this as program MAIN. Build the project as a DLL.

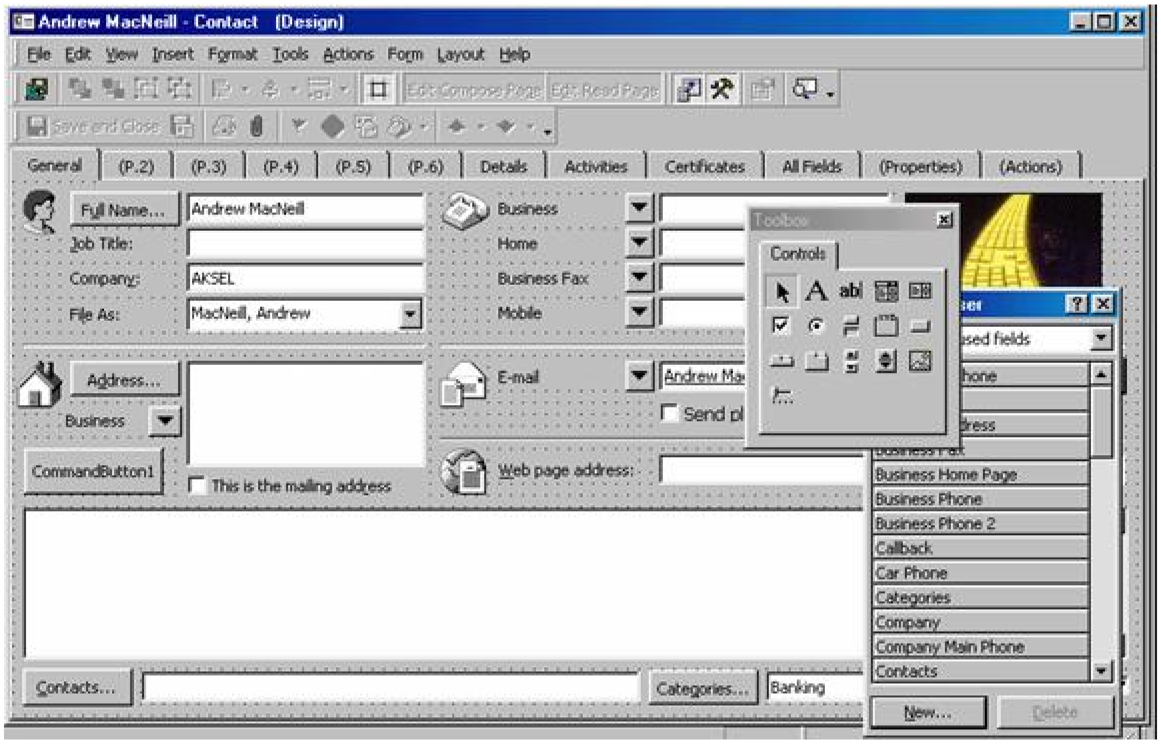
To test the code, try the following code:

```
X = createobject("Outcome.OutTrack")
x.WriteLog("My Outlook automation tracker")
```

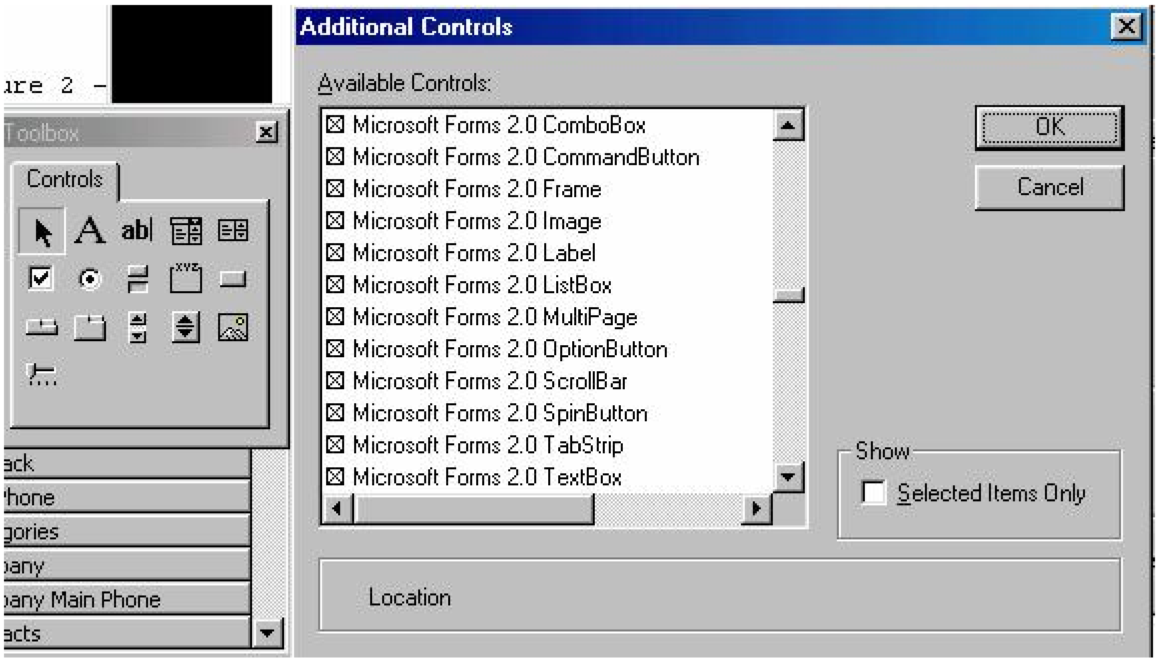
Building the Customized Form

Our next step is to build the Customized Form.

Under the Tools menu, select Forms->Design Form. The Outlook form designer will appear.



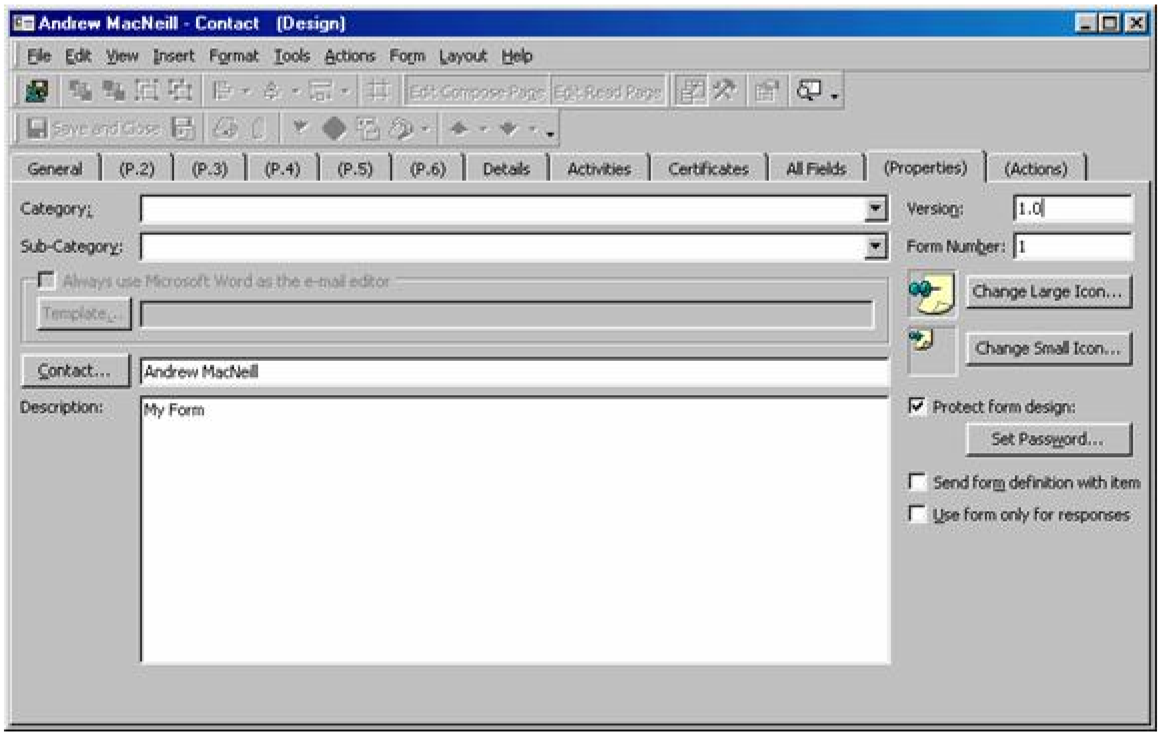
Outlook has a number of built-in controls that are similar to their Visual FoxPro counterparts. The Outlook controls are also available to you in Visual FoxPro! They are the ActiveX controls named Microsoft Forms. Right-click on the Control Toolbox and select Custom Controls to add your own favorite ActiveX controls, such as a ListView or TreeView.


Toolbox
Controls
Additional Controls
Available Controls:
Microsoft Forms 2.0 ComboBox
Microsoft Forms 2.0 CommandButton
Microsoft Forms 2.0 Frame
Microsoft Forms 2.0 Image
Microsoft Forms 2.0 Label
Microsoft Forms 2.0 ListBox
Microsoft Forms 2.0 MultiPage
Microsoft Forms 2.0 OptionButton
Microsoft Forms 2.0 ScrollBar
Microsoft Forms 2.0 SpinButton
Microsoft Forms 2.0 TabStrip
Microsoft Forms 2.0 TextBox
OK
Cancel
Show
Selected Items Only
Location

When you right-click on a control in the Outlook form designer, there are two menu options named Properties. The first one displays a dialog with only the basic properties on it. The second one, named Advanced Properties, displays a Property Sheet where changes may be made to all of the properties. The standard Properties dialog has three tabs for setting basic properties like the name and position as well as providing basic validation.

Clicking the Choose Field button displays a list of "grouped" properties (such as Commonly used fields). Many of the controls in an Outlook form have automatic features. For example, the Categories button in the Contact form displays a dialog where users may identify the categories for a contact. Clicking the Choose Field button for a Command button displays a list of the dialogs that may be called when the user clicks that button. Along with Categories, you can display the Confirm Address or Name dialog for Contacts or the Add Recipients for mail messages.

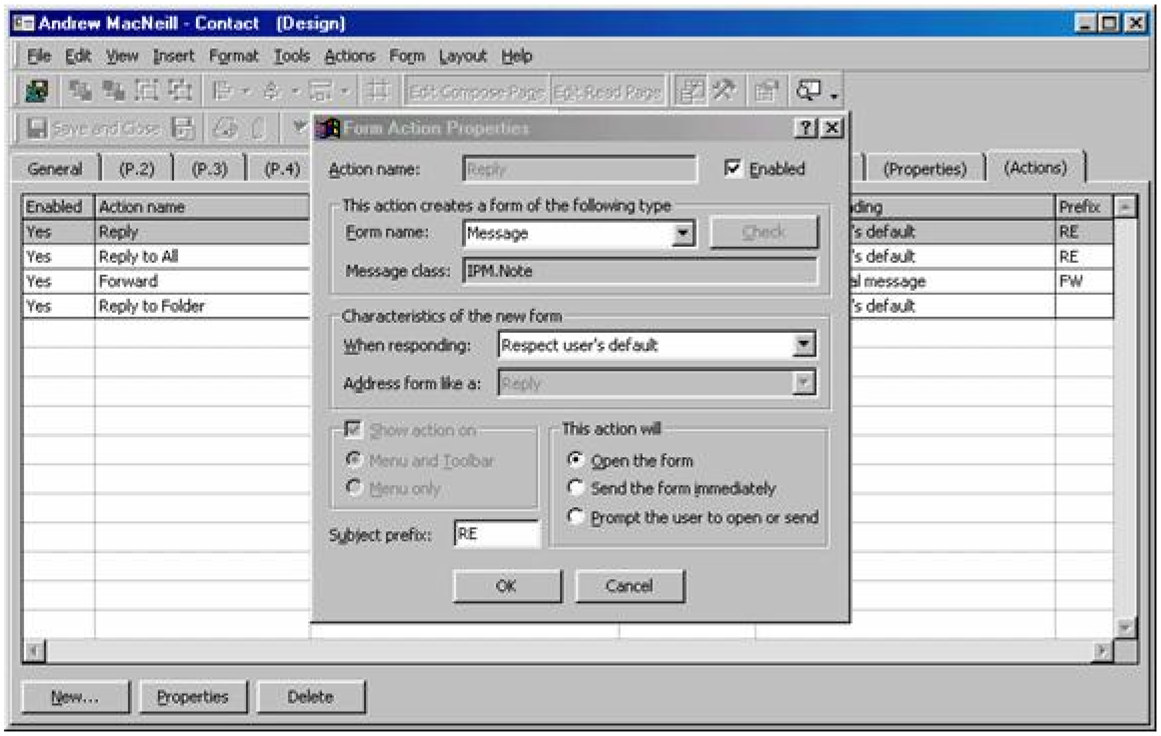
Once the custom form is designed, it needs to be saved. Unlike applications, customized forms cannot be compiled to hide their logic. To protect the design from being changed by others, click on the Properties Tab (see figure 5). Check the box marked Protect Form Design and set the password. This ensures no one will be making changes except you.


Andrew MacNeill - Contact (Design)

Version: 1.0
Form Number: 1
Contact... Andrew MacNeill
Description: My Form
☑ Protect form design:
☐ Send form definition with item
☐ Use form only for responses

The Properties tab in the Form Designer makes it easy to add version and protection to your new form.

Control the related forms users see by changing the settings on the Actions tab. You can also create custom actions that appear on the menu to display other forms and information.


Andrew MacNeill - Contact (Design)
File Edit View Insert Format Tools Actions Form Layout Help
Edit Compose Page Edit Read Page
Save and Close
General (P.2) (P.3) (P.4) (Properties) (Actions)
Form Action Properties
OK Cancel
New... Properties Delete

Enabled	Action name	ding	Prefix
Yes	Reply	's default	RE
Yes	Reply to All	's default	RE
Yes	Forward	l message	FW
Yes	Reply to Folder	's default	

Action name: Reply [✓ Enabled]

This action creates a form of the following type
Form name: Message
Message class: IPM.Note

Characteristics of the new form
When responding: Respect user's default
Address form like a: Reply

[✓ Show action on]
[✓ Menu and Toolbar] Menu only

This action will
[✓ Open the form] Send the form immediately Prompt the user to open or send

Subject prefix: RE

Publishing the Form

Select Save from the File menu to save the form for the currently selected item only. To make the form available for other Outlook items, select Publish Form from the Forms submenu on the Tools menu.

Forms may be published in individual folders or to a Personal Forms Library. When published to the individual folders, the forms will be automatically installed for anyone who opens the form. In the Personal Forms library, they are available only to you.

Customizing Outlook Events

Using forms is only one way of customizing the user's Outlook experience. It can also be time-consuming. Designing individual forms is not the way to reduce your development time. Instead, consider changing the way Outlook reacts behind the scenes.

To do this in Visual FoxPro 6.0 (and 5.0), we need to use Visual Basic to build the initial application that will call Outlook. This application also informs Outlook to use its special methods. These methods (in the Visual Basic application) will call our COM server to do the real work.

In Visual Basic, create a new Project. This new project needs to reference the Microsoft Outlook 9.0 Object Library. Select References from the Project menu and select the library. This tells VB that this object library is part of the application.

Then in either the form or the main subroutine, place the following code:

```
Public WithEvents myOlApp As Outlook.Application
Public Tracker
Private Sub Form_Load()
Set myOlApp = New Outlook.Application
Set Tracker = CreateObject("Outcome.outtrack")
End Sub
Private Sub myOlApp_ItemSend(ByVal Item As Object, Cancel As Boolean)
```

```
' Item.Body = Item.Body & Chr(13) & "Message updated by My Test
Application"
Tracker.WriteLog ("Item Sent: " & Item.Body)
'better version
If Tracker.SendItem(Item) = False Then

    Cancel = True
End If

End Sub
```

The first part of this code

```
Public WithEvents myOlApp As Outlook.Application
```

creates an object named myOlapp which is defined as the Outlook application (similar to VFP's CreateObject statement). The WithEvents clause is crucial here because this notifies Outlook that the VB application has updated Outlook events in it.

The event method myolapp_ItemSend is a direct copy of the ItemSend method we call when attempting to send a message. This method is passed an Item and a Cancel property. If the method code sets the Cancel variable to True, then the Outlook item will not be sent.

In order to use this application, we actually need to instantiate the objects. This code is usually placed at the first call of the program:

```
Set myOlApp = New Outlook.Application
Set Tracker = CreateObject("Outcome.outtrack")
```

Following the Process

When deploying this application, all that is needed is to replace the user's Outlook icon with the VB application icon. After that, the application runs as follows:

When the user starts the application, Outlook is started.

Whenever the user sends an item, the VB method is then executed.

The VB method calls our VFP server, which then logs the information and performs additional functions.

Next Steps

In the above example, we simply added code for sending an item. In a larger application, we might track individual items including saving contacts, scheduling appointments and more.

Uses in Visual FoxPro 7

All of the code in this session could be done in either VFP 6 or 7. VFP 7 introduces use to some more powerful functions. IMPLEMENTS and EVENT HANDLER.

Implements is great except that you have to provide the definitions for all of the implemented Methods. Note that there can be a LOT of methods in this code.

```
DEFINE CLASS oOutlook as Custom
   IMPLEMENTS Application in "Outlook.application"
PROCEDURE application_Get_session
PROCEDURE application_Get_application
PROCEDURE application_Get_answerwizard
PROCEDURE application_Get_class
PROCEDURE application_Get_assistant
PROCEDURE application_Get_comaddins
. . .
PROCEDURE Application_CreateItem (ItemType) as Object
PROCEDURE Application_CreateItemFromTemplate (tmppath,Folder) as object
PROCEDURE Application_CreateObject (objName) as object
pvarResult As Variant, pexcepinfo As EXCEPINFO, puArgErr As Numeric)

ENDDEFINE

Lo = Createobject("Outlook.application")
Los = CREATEOBJECT("oOutlook")
? EVENTHANDLER(lo, los)
```

Please note that the above code works for COM objects such as recordsets and it should also work with Visual FoxPro 7 based on the beta documentation.

Conclusion

One of the goals of this session is to open developers' eyes to the possibilities of using other applications in conjunction with Visual FoxPro. As we have seen, there are a number of different ways to accomplish it. Our application can control what the user does or we can have the user control it, ensuring our application still does exactly what it needs to do.

As you build your solutions, drop me a line and show me what you've done : Andrew@aksel.com.