CHAPTER 11

Communications Programming

Visual Basic is known as a useful tool for prototyping, for developing client-server applications and for general rapid application development. But what are its capabilities when it comes to creating communication applications? Can Visual Basic be considered a serious tool for communication programming? The answer is yes. With its MSComm custom control and support for DDE, OLE and MAPI, Visual Basic provides a well rounded suite of communication tools. In this chapter, we'll be looking at the variety of applications which demonstrate the various communication capabilities that Visual Basic possesses.

The examples in this chapter demonstrate:

- Developing communication programs using the MSComm control.
- Using DDE to access Windows applications.
- Creating DDE server applets.
- Making use of NetDDE.
- Tapping the capabilities of MAPI.

Communication Applications with Visual Basic

In the beginning, Visual Basic was viewed as a tool for developing simple Windows applications. Typically, these were applications with a very limited scope. A project would involve a few forms, a module and a couple of custom controls. Visual Basic was the tool that allowed the novice to create Windows programs.

As Visual Basic matured, additional functionality was provided. Third parties, eager to profit from the success of VB, created a wide spectrum of custom controls. Microsoft has extended Visual Basic's capabilities with every release since and Visual Basic can now be used to produce large, feature-rich Windows applications.

As Visual Basic has gained acceptance as a development language for a variety of business applications, one of its most powerful features has received little attention: its communication capabilities. Visual Basic offers a full set of tools which can be used for developing applications that make use of various types of communication functionality. This includes:

- Serial port communication, either with or without a modem.
- Interapplication communication using DDE or OLE.
- Network communication to another application using NetDDE.
- Electronic mail by interfacing with MAPI compliant mail systems.

In this chapter, we will be looking at all of these capabilities with the exception of OLE which is covered in-depth elsewhere in this book.

Serial Port Communications

Included with Visual Basic Professional Edition is the communications custom control, MSComm. This control provides you with an easy way to utilize the serial ports of your PC. While it's simple to use, it's also powerful.

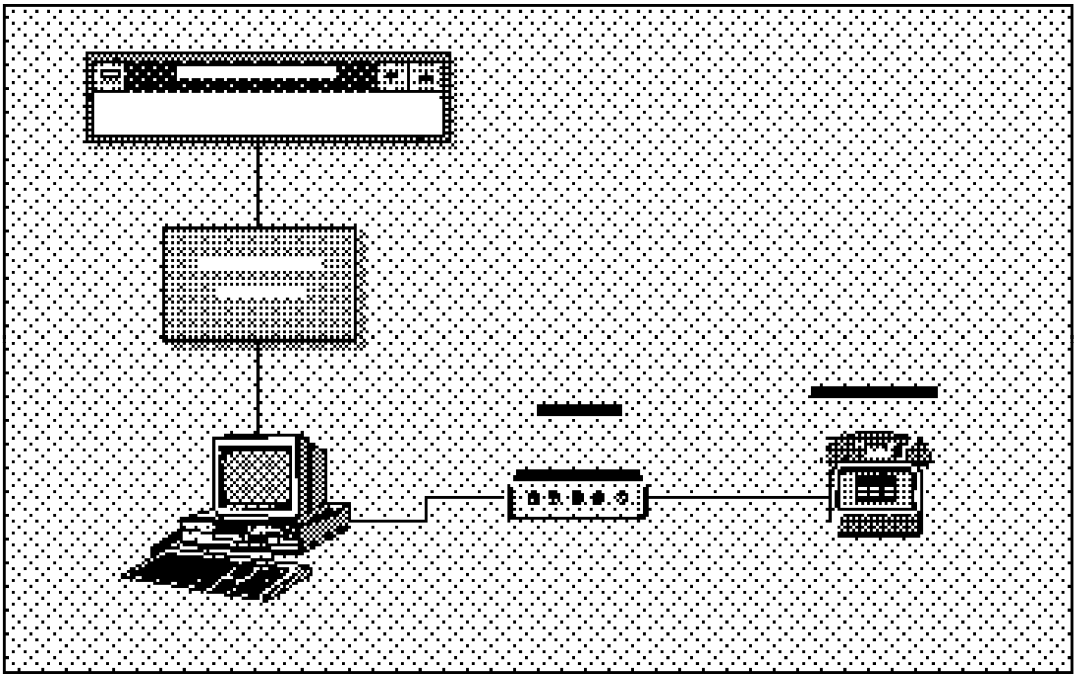
With the MSComm control, you can implement serial communication in two ways:

- By manually polling serial ports.
- By responding to communication events.

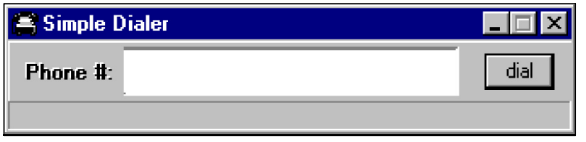
To demonstrate the use of the MSComm control, we'll look at an example program which implements a simple phone dialer. While this application is limited in functionality, it demonstrates the primary functions of the MSComm control.

A Simple Phone Dialer

The following diagram shows where the MSComm control fits in to the picture.



You'll find the example as **Simpdial.vbp** from the **\Chap11\Dial_Ocx** subdirectory on the disc. Use the project window to display **frm_Simple_Dialer**. On this form, you'll see the text box which will be used to accept the phone number to dial and the command button labeled Dial which starts dialing the phone. The communication control is visible only at design-time.



Configuring the Dialer

Before you run this application, you'll probably need to make two modifications in the **Form_Load** procedure. The MSComm control has two properties, one of which specifies a serial port and the other, the settings of that port. You may need to change these lines, as shown below, to match your system configuration:

```
Private Sub Form_Load()

  com_Phone.CommPort = 3
  com_Phone.Settings = "19200,N,8,1"

End Sub
```

The **CommPort** property needs to be set to match the port to which your modem is connected. The **Settings** property is a string that should be set to match the baud rate, parity, data bits and stop bits as needed by your modem. Each of these should be separated by a comma, as you can see from the code.

In a fully functional application, you would want to handle this configuration in a more appropriate manner. For our purposes, though, this method provides a better understanding of how the MSComm control functions.

Once you have set the application running, type in a phone number to dial, including any necessary prefix and click on the Dial command button. Your phone number will be dialed while the status bar is updated with the progress of the call.

What's Happening?

Using the MSComm custom control involves two steps:

1. Configuring the control
2. Writing to and reading from the control.

Configuration of the control is handled in the **Form_Load** and **Dial_Phone** procedures. The MSComm properties could be directly set at design time, but it's more typical to configure them at run-time, as this allows the application greater flexibility for different combinations of computers and modems.

The actual operation of dialing the phone is initiated in the click procedure for the Dial command button. The procedure begins by verifying that something has been entered in the phone number text box. This is a very simple check and doesn't begin to address the possible options for verifying a number. You could modify this application to accept only numbers and certain punctuation, or make use of characters (as in 1-800-GET-WROX) or other combinations. For our purposes, a simple check for existence will suffice.

```
Private Sub cmd_Dial_Click()
  Dim dialing_mode As String * 1
  Dim number_to_dial As String
  Dim response As Integer
  Dim time_to_wait As Integer

' Check to see if a phone number has been entered.
  If (txt_Phone_Number.Text = "") Then
    response = MsgBox("No phone number was provided with dialing action."↵
        , 48, "Dialing Error")
```

```
      Exit Sub
    End If
```

A common routine **Dial_Phone** is called to perform the physical dialing. It makes use of five parameters:

- Number to dial.
- The MSComm control to use.
- Mode of dialing, either 'T' for touch-tone or 'P' for pulse.
- A control on the calling application's form which can be used to display the status of the dialing operation.
- The time to wait in seconds for the modem to respond to the dialing operation.

```
' Define the parameters to use when dialing the phone.
  dialing_mode = "T"
  number_to_dial = txt_Phone_Number.Text
  time_to_wait = 155

' Initiate the dial.
  Call Dial_Phone(number_to_dial, com_Phone, dialing_mode, pn3_Status,↵
         time_to_wait)

End Sub
```

The **Dial_Phone** procedure is designed as a common routine which can be included into any project that requires simple phone dialing capabilities.

```
Sub Dial_Phone(phone_number As String, com_control As Control,↵
         dialing_mode As String, status_control As Control,↵
         time_to_wait As Integer)
' In addition to supplying the two controls mentioned above an object that
' is part of your UI needs to have its click event contain the following
' line:
'
'    dialing_in_process = False
'
' This will be used to disconnect the modem once the phone has been dialed
' and the user has picked up the phone.

  Const COMMA = 44
  Const NUMBER_0 = 48
  Const NUMBER_9 = 57

  Dim beginning_time As Long
  Dim cnt As Integer
  Dim current_time As Long
  Dim modem_command_string As String
  Dim no_punct_phone_number As String
  Dim response_from_modem As String
```

It begins by removing any characters in the phone number that are neither the numbers 0 through 9 or a comma, which is the Hayes command for pause. You'll find that the pause character will have additional uses as you create more complex phone dialers.

```
' Build string to perform the "dialing". First remove any unwanted
' characters (anything but 0-9 and "," - the Hayes pause character.).
  no_punct_phone_number = ""
  For cnt = 1 To Len(phone_number)
    Select Case Asc(Mid$(phone_number, cnt, 1))

' Current character is a number.
      Case NUMBER_0 To NUMBER_9
        no_punct_phone_number = no_punct_phone_number & Mid$(phone_number
        , cnt, 1)

' Current character is a comma.
      Case COMMA
        no_punct_phone_number = no_punct_phone_number & Mid$(phone_number
        , cnt, 1)

' Anything else is ignored.
    End Select
  Next cnt
```

After all unwanted characters have been removed from the phone number string, it's combined with the Hayes command for dialing a number. Here the parameter **dialing_mode** is concatenated with the dialing command.

```
modem_command_string = "ATD" & dialing_mode & no_punct_phone_number & ";"
         & vbCr
```

With all of the preparation completed, the serial port is opened using the MSComm property **PortOpen**. This property is used to set and return the state of a COM port. We use an **OnError** statement in conjunction with this command to trap any problems encountered while opening the port.

```
' Open the COM port.
  On Error Resume Next
  status_control = "Opening COM port..."
  com_control.PortOpen = True
  If (Err) Then
    status_control = "Error opening port. Please check modem and
         configurations."
    Exit Sub
  End If
```

You communicate with the MSComm control through the use of the **Input** and **Output** properties. Here we start by resetting the **InBufferCount** property to 0. This property contains the number of characters waiting in the receive buffer. By setting **InBufferCount** to 0 you are, in effect, clearing the receive buffer.

```
com_control.InBufferCount = 0
```

Finally, we're ready to perform the physical dialing of the phone. To accomplish this, we load our dialing command string into the **Output** property. This results in the string being written to the transmit buffer of the MSComm control.

```
' Dial the phone.
  status_control = "Dialing number..."
  com_control.Output = modem_command_string
```

After the dialing command has been transmitted to the modem via the communication control, we wait for a receipt of message signal. Hayes-compatible modems respond with **"OK"**.

Polling vs Events

In the next section, we'll look at one of the two ways to interact with the MSComm control. At the start of the chapter, we noted that MSComm provides two methods for handling communication:

- Polling the control either continuously, on regular intervals, or at critical times in your application.
- Through the use of the MSComm control's **OnComm** event.

In this example, we're using the polling method, due to the simplicity of the application. In more complex communication programs, you may find that the use of the **OnComm** event is more suitable.

To poll the communication control, we simply loop and check the **InBufferCount** property. If there's anything in the buffer, it's added to a response string. This string in turn is checked for the **"OK"** message.

```
' Check for an "OK" (literally) back from the modem.
' This is the expected response in this situation. We will loop until
' this occurs.
  dialing_in_process = True
  Call Get_Time_In_Seconds(beginning_time)

  Do While dialing_in_process
    DoEvents
    If (com_control.InBufferCount > 0) Then
      response_from_modem = response_from_modem + com_control.Input
      If (InStr(response_from_modem, "OK")) Then
        Beep
        status_control = "Dialing completed. Pick up phone and click on↵
         form."
```

After the phone has been successfully dialed and the call is being processed, we need a method to transfer the call from this application's control to the user. This is accomplished through the use of the global variable **dialing_in_process** and a click event. The process that is occurring here is:

1. The modem dials the phone.
2. Modem waits for an OK response.
3. When it receives the OK, the modem signal causes the program to inform the user that the line is up through a status message.
4. Modem waits for the user to acknowledge that they have picked up the phone's receiver by clicking on the form. This closes down the modem.

To signal periods when the application is ready to receive information from the user, we add the following line of code to the click event of an object of our demo application:

```
dialing_in_process = False
```

When the user clicks on the form, the global variable is set to **False**, so the following simple loop in the **Dial_Phone** procedure completes, as does the larger dialing loop.

```
        ' When we get here, the phone has been dialed and the modem has
        ' acknowledged that it was completed. This little loop simply waits
        ' for the user to pick up the phone and click on the form.
          Do While dialing_in_process
            DoEvents
          Loop
        End If
      End If
```

At the same time as we are monitoring the input buffer for a response, we monitor the time that has elapsed since the dialing command was initiated. If we've waited longer than the **time_to_wait** parameter, then the dialing operation is aborted.

```
      Call Get_Time_In_Seconds(current_time)
      If (current_time > beginning_time + time_to_wait) Then
        status_control = "Modem not responding. Dialing aborted."
        com_control.PortOpen = False
        Exit Sub
      End If
    Loop
```

Shutting Down

The final function to perform when control is transferred from our application to the user, is to disconnect from the modem and close the COM port. Sending the Hayes command **"ATH;"** causes the modem to disconnect. The port is closed by setting the **PortOpen** property of the MSComm control to **False**.

```
  ' Disconnect the modem.
    status_control = "Disconnecting from phone..."
    com_control.Output = "ATH;" + vbCr

  ' Check for an "OK" or "ERROR" (literally) back from the modem. This is
  ' the expected response in this situation. We will loop until this
  ' occurs.
    response_from_modem = ""
    Do While (InStr(response_from_modem, "OK") = 0) And↵
           (InStr(response_from_modem, "ERROR") = 0)
      response_from_modem = response_from_modem & com_control.Input
      DoEvents
    Loop
    status_control = "Disconnected from phone..."
    DoEvents

  ' Close the COM port.
    com_control.PortOpen = False

    If unload_the_form Then
      Unload com_control.Parent
    End If

  End Sub
```

The user may try to unload the form while execution is in the loop that tries to dial the number. The code in this procedure references controls on the form, meaning that if the form is unloaded, it will be reloaded again. We need to make sure that this procedure has a method of unloading the form, or else the program will never end. We achieve this using the global flag **unload_the_form** and the following code in the **QueryUnload** event of the form:

```
Private Sub Form_QueryUnload(Cancel As Integer, UnloadMode As Integer)
' If we are trying to dial, set the flags so that the loop
' will end and the form will be unloaded by the code in mod_Dialing
  If dialing_in_process Then
    If UnloadMode <> vbFormCode Then
      unload_the_form = True
      dialing_in_process = False
      Cancel = True
    End If
  End If

End Sub
```

Summary

While this is a very simple example of using the MSComm control, it demonstrates its basic functionality and use. Several points that are critical parts of a successful communication application that were shown here are:

- Communications programs can be affected by numerous extraneous situations. Error handling is a critical part of all successful applications using the MSComm control.
- Since much of the interaction with the communication control is hidden from the user, you must incorporate a method for updating status information for the user as a communication process progresses.
- If polling techniques are used by your application, you must allow time for other events to occur by strategic use of the **DoEvents** command.

Communicating with DDE

Modem-based system-to-system communications isn't the only sort of communication necessary in a modern computing environment. Windows and its multitasking environment make it possible to communicate between applications. We've already extensively covered one method for this communication: OLE. In this section we'll examine the other major method: **dynamic data exchange**, or DDE.

With DDE you can link applications, allowing the transfer of data. DDE lets you establish a **conversation** between two applications. The application that initiates the conversation is referred to as the **destination**. The target of the DDE conversation is referred to as the **source**. Although, as we'll see, it does allow for two way communication, DDE is primarily a method of requesting information. Data tends to flow from source to destination.

To establish a DDE conversation, the destination application must specify:

- The name of the source application: an example would be Word, Excel or even another Visual Basic program.
- The topic of the conversation such as a document, spreadsheet or object relating to the source application.
- The item of the conversation: this could be a bookmark, spreadsheet cell or control.

- The type of conversation: these are commonly referred to as **links**. There are three types of links: **automatic**, **manual** and **notify**. With an automatic link, the destination application is updated automatically whenever the item changes on the source. In manual link mode, the source application only sends an update when requested by the destination application. With a notify link, the source notifies the destination whenever an item changes but does not send the value unless asked.

You've already seen a glimpse of DDE in action when we used it to bring up the Find File dialog from the drag-and-drop example in Chapter 4. In the following sections, you'll see several more examples which use DDE from Visual Basic.

DDE_Dial: A Simple Use of DDE

DDE_Dial is a simple example that demonstrates how Visual Basic can be used to create both a server and destination application. You'll find the code for this example in the **\Chap11\DDE_Dial** subdirectory on the disc.

We'll begin by looking at the server application, **DDE_Dial.vbp**. This program makes use of the same dialing routines presented earlier in this chapter to provide a simple phone dialer which can be activated via DDE. In fact, **DDE_Dial.vbp** is almost exactly the same as **SimpDial.vbp** that you saw earlier, except for three important differences:

- The **LinkMode** property of the main form was set to Source
- The **LinkTopic** property of the main form was set to **frm_DDE_Dial**.
- Some code was added to the **txt_Phone_Number_Change** event procedure so that the number would be dialed whenever the text in the phone number box was changed.

> **In this new incarnation, I've made the form smaller. If you enlarge it again, you'll see that it contains the same controls as the original.**

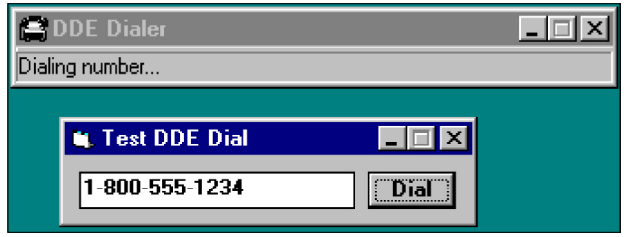
As with the previous dialing example, before you run this application you'll probably need to make modifications to the **CommPort** and **Settings** properties used. The settings are located in the **Form_Load** section of **frm_DDE_Dial**. You may need to change the lines shown below to match your system configuration:

```
com_Phone.CommPort = 2
com_Phone.Settings = "2400,n,8,1"
```

Testing DDE_Dial

To test DDE_Dial, we must create a client application, **Testdial.vbp** . The test program will accept a phone number from the user and, when instructed, will transfer the number to DDE_Dial for processing.

If you run the program, you'll see that both the test client and the server application have been started. Enter a number in the text box on the client form and click the Dial button. The number will be sent via DDE from the client to DDE_Dial, which will process the request and notify you of dialing progress through the panel on the server application.



What's Happening?

When the test program starts, it establishes a DDE link with DDE_Dial in its **Form_Load** procedure. We link the text box control, **txt_Phone_Number**, in the test application to the text box control (coincidentally also called **txt_Phone_Number**) in the server application. By setting its **LinkTopic**, **LinkItem** and **LinkMode**, a DDE link is created between itself and DDE_Dial.

The value used to set the **LinkTopic** property should be in the form: *source application* plus the pipe character "|", followed by the *data group in the source application*. In this case, the application name is the name of the project, **DDEDIAL**, and the data group name is the **LinkTopic** property of the source form, **frm_DDE_Dial**.

The **LinkItem** property is set equal to the name of the text box control on the source application. It should be noted that even though the text box controls on both the source and destination applications have the same name, **txt_Phone_Number**, in the code below we are referring to the control on the source application.

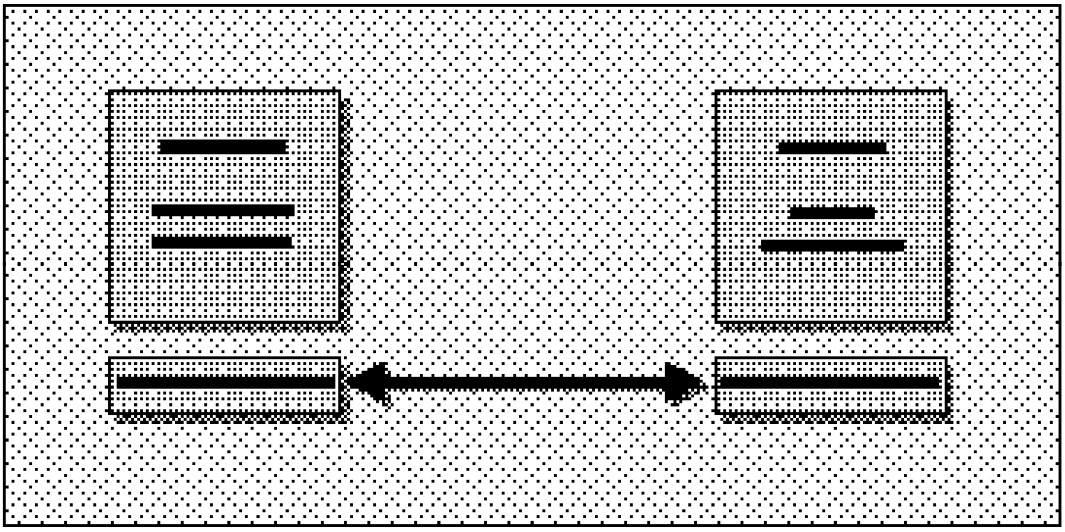
LinkMode refers to the mode of operation that will be used with this DDE conversation. In this example, we've used the manual mode. This means that no data will be transferred between applications unless requested.

```
Private Sub Form_Load()
  Const DDE_NO_APP = 282
  Dim return_value As Long

' Establish link with DDE dialer.
  On Error GoTo Start_DDE_Dial
  txt_Phone_Number.LinkMode = vbLinkNone
  txt_Phone_Number.LinkTopic = "DDEDIAL|frm_DDE_Dial"
  txt_Phone_Number.LinkItem = "txt_Phone_Number"
  txt_Phone_Number.LinkMode = vbLinkManual
  On Error GoTo 0
  Exit Sub
```

For a DDE conversation to be established, the source application must be running. While there are several methods for handling this situation, I believe that the one I've used here works best. By setting an error handler at the top of the **Form_Load** procedure, we'll trap errors that occur as a result of the absence of the server application. In the error handler we check the cause of the error. If it's the result of the server not being available, the **Shell** statement is used to launch the server. For any DDE applications you create, you will need to perform something similar to this to verify that your server is accessible.

```
' Error handle for use during establishing of link.
' If error resulted from the source app not currently
' running, start the source app.
Start_DDE_Dial:
  If (Err = DDE_NO_APP) Then
    return_value = Shell(App.Path & "\Ddedial.exe", 1)
    Resume
  Else
    MsgBox Error(Err)
    End
  End If
End Sub
```

When the **Form_Load** procedure is executed a link is established between the text control **txt_Phone_Number** on **frm_Test_Dial** and the text control **txt_Phone_Number** on the form **frm_DDE_Dial** as shown in the diagram below:



When the user enters a phone number and clicks the Dial command button, the following procedure is executed. Using the **LinkPoke** method the destination application transfers the value in its **txt_Phone_Number** control to the source application's **txt_Phone_Number** control. The change event procedure for the source application's text control is then executed and the number is dialed.

```
Private Sub cmd_Dial_Click()
  On Error Resume Next
  txt_Phone_Number.LinkPoke
End Sub
```

You can see that using the **LinkPoke** method makes it possible to transfer data from the destination to the source.

Summary

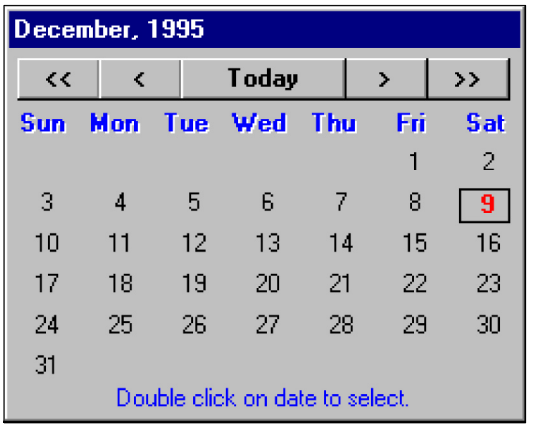
While this is a very simple example of using Visual Basic to create both a DDE source and destination program, it's demonstrated the principle steps involved in a DDE conversation.

In the next example, we'll look at another set of DDE applications where both the source and destination programs are created using Visual Basic. The DDE conversation will be more elaborate, with data being transferred in both directions between the two applications.

DDECalendar

The previous example demonstrated how you can use DDE to create an applet, a single purpose program which can be incorporated into a wide spectrum of applications. In this example, we'll create another small utility - a date selection utility which provides a calendar from which the user can pick a date. This utility will accept and return dates using DDE, and you can use it with any application which can function as a DDE destination.

You can find this example in the **\Chap11\DDECalen** subdirectory on the disk in the form of the two projects **Calendar.vbp** and **Demo.vbp**. To see the server in action, load and run **Calendar.vbp**. It should appear as shown below, although the month and date will vary, depending upon your PC's clock.



You can move between months using the single arrow buttons. The double arrow buttons move forward and back a year at a time. Dates are selected by double-clicking.

DDECalendar, like DDE_Dial is a DDE server application. As such, it isn't designed for use independently of a destination application. While DDECalendar at least has a user interface which can be manipulated, it's still designed to work in conjunction with other programs.

What's Happening?

The calendar program begins by performing some standard initialization activities. The form is centered and tooltips are enabled. Under 16-bit Windows, a call to the procedure **Make_System_Modal** defines the calendar program as the current system modal window. This restricts the user to working only with the calendar until a date is selected. A call to **Build_Calendar** configures the calendar's interface to a specific month.

```
Private Sub Form_Load()

' Setup the form before we begin.
  Center_Form Me
  Load frm_ToolTip
  #If Win16 Then
    Call Make_System_Modal(Me.hWnd)
  #End If

' Display calendar for todays date.
  current_date = Now
  Call Build_Calendar(current_date, current_day, current_month,↵
         current_year, days_in_month, day_of_first)

End Sub
```

The **Make_System_Modal** procedure uses an API call, located in the file **mod_Calendar**, to set the specified window as system modal. The declaration is in the **(General)(declarations)** section of that file.

```
Option Explicit
#If Win16 Then
  Declare Function SetSysModalWindow Lib "User" (ByVal hWnd As Integer)↵
```

```
          As Integer
#End If

Sub Make_System_Modal(ByVal form_hWnd As Integer)
  Dim return_value As Integer
  #If Win16 Then
    return_value = SetSysModalWindow(form_hWnd)
  #End If
End Sub
```

The function just takes the handle of the window to make system modal as its only parameter.

Build_Calendar

This procedure configures the calendar interface to represent a specific month. It performs date calculations to determine the number of days in a month and the first date of the month.

```
Sub Build_Calendar(current_date As Date, current_day As Integer,↵
          current_month As Integer, current_year As Integer,↵
          days_in_month As Integer, day_of_first As Integer)
  Dim calendar_index As Integer
  Dim first_of_month As Variant
  Static previous_day As Integer
  Static previous_month As Integer
  Static previous_year As Integer

' Get numeric values for the date passed.
  current_day = Day(current_date)
  current_month = Month(current_date)
  current_year = Year(current_date)
```

At the beginning of the procedure, a check is made to determine if the specified month is already displayed. If so, there's no reason to regenerate the month.

```
If (current_month <> previous_month) Or (current_year <> previous_year)↵
          Then
```

If either the year or month have changed, then we need to redisplay the month on the calendar. We begin by showing the month and year in the calendar's caption. Then a simple **SelectCase** statement determines the number of days in the current month.

```
  frm_Calendar.Caption = Format(current_date, "mmmm, yyyy")

' Determine the number of days in the current month.
    Select Case current_month
      Case 1, 3, 5, 7, 8, 10, 12
        days_in_month = 31
      Case 4, 6, 9, 11
        days_in_month = 30
      Case 2
        If ((current_year Mod 4) = 0) Then
          days_in_month = 29
        Else
          days_in_month = 28
        End If
    End Select
```

The day of the week on which the first day of the month falls is determined by using two Visual Basic functions: **DateSerial** and **Weekday**. **DateSerial** takes three arguments year, month and day and returns a date that can be stored in a variable of Variant data type or the new Date data type introduced with VB4. Once we've obtained a date for the first of the current month, we can pass this to the **Weekday** function which returns the day of the week that this date falls on.

```
' Determine day of week for first of month.
    first_of_month = DateSerial(current_year, current_month, 1)
    day_of_first = WeekDay(first_of_month)
```

Finally, we're ready to display the new calendar. First, the existing calendar is cleared. Then the new dates are loaded into the captions of the labels.

```
' Clear the days currently on the calendar.
    For calendar_index = 0 To 36
      With frm_Calendar
        .lbl_Day(calendar_index).Caption = ""
        .lbl_Day(calendar_index).BorderStyle = 0
        .lbl_Day(calendar_index).FontBold = False
        .lbl_Day(calendar_index).ForeColor = vbBlack
      End With
    Next calendar_index

' Load the days for the current month.
    For calendar_index = 1 To days_in_month
      frm_Calendar.lbl_Day(calendar_index + day_of_first - 2).Caption↵
          = Str(calendar_index)
    Next calendar_index
```

The current date is saved to use as a comparison the next time this routine is called. This keeps the calendar display from being updated unnecessarily.

```
' Save the day, month, year combinations for next time.
        previous_day = current_day
        previous_month = current_month
        previous_year = current_year
```

The following **Else** branch is used when the calendar display already contains the correct month and year and the user just changes the day. In this case, all that has to be done is to clear the currently highlighted date.

```
Else

' Clear the current highlighted day.
    With frm_Calendar
      .lbl_Day(previous_day + day_of_first - 2).BorderStyle = 0
      .lbl_Day(previous_day + day_of_first - 2).FontBold = False
      .lbl_Day(previous_day + day_of_first - 2).ForeColor = vbBlack
    End With

' Save the day for the next time.
    previous_day = current_day

End If
```

Now, whichever branch the program took, all that remains is to highlight the current day.

```
' Highlight the current day.
  With frm_Calendar
    .lbl_Day(current_day + day_of_first - 2).BorderStyle = 1
    .lbl_Day(current_day + day_of_first - 2).FontBold = True
    .lbl_Day(current_day + day_of_first - 2).ForeColor = vbRed
  End With
End Sub
```

Controlling Movement Between Dates

Whether the user chooses to change the selected date by clicking on it, or by using the command buttons at the top of the calendar, the code uses the same VBA function **DateAdd** to change the current date.

DateAdd allows you to perform 'date math'. It adds the specified interval of time to a date and returns the new date. The first parameter determines what interval to use (days months, years, etc.); the second parameter determines how many of the interval to add; and the last parameter is the date which you wish to add things to. It makes the job of moving between months, years and days very simple. Once the new date has been found, the calendar is refreshed by a call to the **Build_Calendar** procedure which we examined earlier.

Here you can see the code for the click events of the label array that makes up the calendar and the command buttons that alter the year and month. The click events for the Previous Year and Previous Month buttons are identical to those for the Next Year and Next Month buttons except that the second parameter is -1.

```
Private Sub lbl_Day_Click(Index As Integer)
  current_date = DateAdd("y", Index - current_day - day_of_first + 2,↵
         current_date)
  Call Build_Calendar(current_date, current_day, current_month,↵
         current_year, days_in_month, day_of_first)
End Sub

Private Sub cmd_Next_Month_Click()
  txt_Date.SetFocus
  current_date = DateAdd("m", 1, current_date)
  Call Build_Calendar(current_date, current_day, current_month,↵
         current_year, days_in_month, day_of_first)
End Sub

Private Sub cmd_Next_Year_Click()
  txt_Date.SetFocus
  current_date = DateAdd("yyyy", 1, current_date)
  Call Build_Calendar(current_date, current_day, current_month,↵
         current_year, days_in_month, day_of_first)
End Sub
```

The procedure for the Today button simply uses the **Now** function to determine the current date.

```
Private Sub cmd_Today_Click()
  txt_Date.SetFocus
  current_date = Now
  Call Build_Calendar(current_date, current_day, current_month,↵
```

```
            current_year, days_in_month, day_of_first)
    End Sub
```

The DDE Interface

DDECalendar accepts a date to display on startup via DDE. The text box control **txt_Date** (hidden well off the edge of **frm_Calendar**) is used with the DDE link. Anytime that the value of this control changes the calendar is updated. This functionality allows the calendar to be called with an existing start date that may be different from the current date.

```
    Private Sub txt_Date_Change()
      current_date = txt_Date.Text
      Call Build_Calendar(current_date, current_day, current_month,↵
            current_year, days_in_month, day_of_first)
    End Sub
```

The **txt_Date** control is also used to pass the new date to the destination application so when a new date is selected via a double click by the user, we put it into the text box. You'll see how this is used in the following calendar demo program:

```
    Private Sub lbl_Day_DblClick(Index As Integer)
      txt_Date.Text = Format(current_date, "mmmm d, yyyy")
      Unload Me
    End Sub
```

Testing DDECalendar

Testing DDECalendar is handled by a demo destination application, **Demo.vbp**. The test program consists of two text box controls which are used to retrieve a beginning and ending date from DDECalendar.

If you run the program, you'll see the form shown below and you'll also note that the server application, DDECalendar, has been started. Find a date and select it by double clicking. The calendar will be removed and the selected date will be deposited in the Start Date text box on the demo program. Move to the End Date text box. Again, the DDECalendar will be displayed so that you can select another date and the End Date box will be updated.

To demonstrate how dates can be passed in both directions between the demo program and DDECalendar click again on the Start Date text box. Note how the calendar starts up with the initial date set to the date from the Start Date text box

What's Happening?

The only code in **frm_Demo** is contained in the **GotFocus** event procedures for the two text boxes.

```
    Private Sub txt_End_Date_GotFocus()
      Call Get_Date(txt_End_Date)
    End Sub

    Private Sub txt_Start_Date_GotFocus()
      Call Get_Date(txt_Start_Date)
    End Sub
```

When the demo program starts, the **GotFocus** event for the Start Date text box triggers establishment of a connection with DDECalendar. The procedure **Get_Date** handles the DDE communication. This procedure is identical in structure to the DDEDialer demo example with one exception. Here, not only do we send the current date to the source program using **LinkPoke**, but we also set it so that the date will be returned by setting the **LinkMode** to **vbAutomatic**. In this mode, whenever the linked item changes on the source application it will automatically be returned to our destination application.

The process of events that occurs during this DDE conversation is:

1. An attempt is made to establish a conversation with the source application.
2. If the source isn't running, it's started.
3. The current value of the date text box on the destination program is sent to the source program using **LinkPoke**.
4. An automatic link is established between the appropriate destination text box and the **txt_Date** text box control on the source program.
5. On 16-bit systems the source program, being set to system modal, is in control until the user selects a date by double clicking.
6. When the user selects a date it's loaded into the **txt_Date** control on the source application. Since this control is linked to the destination program, the date's transferred to the appropriate text box on the destination application.

```
Sub Get_Date(date_control As Control)
  Const DDE_NO_APP = 282
  Dim return_value As Long

' Load the calendar utility, pass the current value of the destination
' object to the calendar and then wait for the new value to be returned.
  On Error GoTo Start_Calendar
  date_control.LinkMode = vbLinkNone
  date_control.LinkTopic = "Calendar|frm_Calendar"
  date_control.LinkItem = "txt_Date"
  date_control.LinkMode = vbLinkManual
  date_control.LinkPoke
  date_control.LinkMode = vbLinkAutomatic
  Exit Sub
```

This error handler is identical to the previous DDE demo. It's used to handle the situation where the source DDE application is not currently loaded.

```
' An error was encountered trying to access the calendar utility.
' Check the error and handle accordingly.
Start_Calendar:
  If (Err = DDE_NO_APP) Then
    return_value = Shell(App.Path & "\calendar.exe", 1)
    Resume
  Else
    MsgBox Error(Err)
```

```
        End
      End If

    End Sub
```

Conclusion

This example displays how to use the primary functions of DDE to transfer data both ways during a conversation. It shows the differences between the automatic and manual link modes and how they're used to add DDE functionality to a Visual Basic application.

DDE to Excel

In the previous two sections, you were shown examples of how you can use Visual Basic to create both server and destination applications. While this capability in itself can add significant functionality to your programs, DDE can also be used to interface with other applications. This section contains an example which uses DDE with Microsoft Excel.

The task of this example is to create a sign-out log for a car pool. We'll use an Excel workbook containing twelve sheets, one for each month of a year. On each sheet there's a row for each day of the month and a column for one of ten types of vehicles that are available to the employees.

Note this example requires Microsoft Excel to be loaded on your PC.

The Car Request application, **Car_rqst.vbp** found in the **\Chap11\DDEExcel** subdirectory on the disk, uses two principle forms, a splash form and the primary interface form. Unlike the previous two examples, where the server applications were fairly small and loaded quickly, starting Excel can be time consuming so the splash form is intended to pacify the user during this process. To create a splash form that will load quickly you need to keep the number of controls to a minimum.

When Excel has finished loading, the splash form is unloaded and the Car Request form is displayed. DDECalendar is automatically launched when Car Request is loaded so that you can select a date for reserving a car by double clicking on a date. The calendar will be removed and the selected date will be deposited in the Date text box.

Entering the Employee's Name

Before a car can be requested, an employee name must be entered. Typically, this would be a good situation to use a combo box, where the user would be restricted to selecting from a specific list of names, but for the purpose of our example a text box is sufficient.

Specifying Seating and Storage Space

You specify the minimum requirements for your vehicle by using the two combo boxes located at the bottom of the form. Only vehicles which meet your requirements will be available to select from.

Building a List of Vehicles 

Before you can select a car you must build a list of available vehicles. Once you have entered the date, storage and seating requirements, click the Find available cars button to populate the Car drop-down listbox with the cars that are available on your chosen date and which meet the minimum requirements.

Try changing the seating and storage requirements and rebuild the list of available cars. Note how the list varies depending upon the date and your parameters.

Reserving a Car 

When you have built a list of available vehicles based upon your requirements, you are ready to reserve your selection. Pick a vehicle from those available in the drop-down list box and reserve it by clicking the Take car button.

What's Happening?

The Car Request program begins with the loading of the splash form, **frm_Splash**. In the **Form_Load** procedure for that form, we center the form and then force its display. If the form was not explicitly displayed it would not be shown until after the startup procedure was completed, which would defeat the purpose intended for the splash form. The **DoEvents** statement allows time for Windows to display the form.

```
Private Sub Form_Load()
' Display the splash form.
  Center_Form Me
  Me.Show
  DoEvents
```

Various global variables are initialized at this point. If you're running the 16-bit version, you'll probably need to alter **EXCEL_PATH** so that it matches the path to Excel on your system. Otherwise, you should be able to leave everything as it is.

```
' Setup the Location of the Excel Spreadsheet.
  DATA_FILE = "[car_rqst.xls]"
  DATA_FILE_ONLY = App.Path & "\car_rqst.xls"
  #If Win32 Then
    EXCEL_PATH = GetRegistrySetting(HKEY_LOCAL_MACHINE, "Software\↵
        Microsoft\Windows\CurrentVersion\App Paths\Excel.exe", "")
  #ElseIf Win16 Then
  ' Modify this for your own system
    EXCEL_PATH = "C:\Office\Excel\Excel.exe"
  #End If
```

Once the splash form is displayed a check is made to determine whether Excel is running. This procedure will load Excel if necessary, and load the appropriate spreadsheet for our application.

```
  Check_For_Excel

End Sub
```

The procedure **Check_for_Excel**, found in **mod_Car_Request**, performs several functions. It begins by attempting to establish a DDE link to Excel. Using the same error handling techniques as demonstrated in the previous DDE examples, it will trap link errors and start Excel if it isn't running.

```
Sub Check_For_Excel()
  Const DDE_NO_APP = 282
  Dim return_value As Long

' Set up a DDE link to Excel.
  On Error GoTo Start_Excel_Now

  With frm_Splash
    .txt_Link.LinkMode = vbLinkNone
    .txt_Link.LinkTimeout = 10
    .pn3_Status.Caption = " Establishing link to Excel."
    .txt_Link.LinkTopic = "Excel|system"
    .txt_Link.LinkMode = vbLinkManual
    .txt_Link.LinkMode = vbLinkNone
    .txt_Link.LinkTimeout = -1
    .txt_Link.LinkMode = vbLinkManual
  End With
```

Once a link has been established to Excel, the default spreadsheet is closed using the **LinkExecute** method. While this step is not critical to the operation of this example, it demonstrates the usefulness of the method.

```
    On Error GoTo 0
    frm_Splash.txt_Link.LinkExecute "[Close(False)]"
```

The status panel located at the bottom of the splash form keeps the user informed as to the progress of the load procedure. By updating this panel the user knows that events are occurring. We then use the **LinkExecute** method again to open the spreadsheet we use for keeping track of the available cars.

```
On Error GoTo Start_Excel_Now
  With frm_Splash
    .pn3_Status.Caption = " Loading car request spreadsheet."
    .txt_Link.LinkExecute "[Open(""" & DATA_FILE_ONLY & """)]"
    .pn3_Status.Caption = " Excel ready, starting app."
  End With
```

Finally, with Excel running and the appropriate spreadsheet loaded, we're ready to begin. All that's left to do is to remove the splash form and load the Car Request main form.

```
' Everything is ready so remove the splash form and display the main form.
  Unload frm_Splash
  Load frm_Car_Request
  Exit Sub
```

The error handling routine at the bottom of this procedure is identical to those discussed in the previous DDE examples.

```
' An error was encountered trying to access Excel.
' Check the error and handle accordingly.
Start_Excel_Now:
  If (Err = DDE_NO_APP) Then
    frm_Splash.pn3_Status.Caption = " Excel not running. Starting Excel."
```

```
      return_value = Shell(EXCEL_PATH, vbMinimizedNoFocus)
      Resume
    Else
      MsgBox Error(Err)
      End
    End If

End Sub
```

The **Form_Load** procedure for the Car Request form is very simple. It begins by initializing tooltips for the form's toolbar. It then initializes the combo boxes for available cars, seating and storage.

```
Private Sub Form_Load()
  Dim row_index As Integer

' Setup the form before we begin.
  Load frm_ToolTip
  Me.Show
  DoEvents

' Load seating options.
  With cmb_Seating
    .AddItem "2"
    .AddItem "3"
    .AddItem "4"
    .AddItem "5"
    .AddItem "6"
    .AddItem "7"
    .ListIndex = 0
  End With

' Load storage options.
  With cmb_Storage
    .AddItem "Small"
    .AddItem "Medium"
    .AddItem "Large"
    .ListIndex = 0
  End With

' Load available cars options.
  cmb_Car.AddItem "None"
  cmb_Car.ListIndex = 0

End Sub
```

The first control on **frm_Car_Request** is the date text box as it has a **TabIndex** property of zero. The effect of this action is that when the Car Request form is loaded the date text control receives the focus. The **GotFocus** event associated with that text box triggers establishment of a connection with DDECalendar as shown in the previous example. This combination of configurations and events results in a date being solicited when the program is initiated.

```
Private Sub txt_Date_GotFocus()
  DoEvents
  Call Get_Date(txt_Date)
End Sub
```

After a date has been selected, the Car Request form should appear as shown here. This is the first time since initiation of the program that the user has complete control over the application. Up to this point the program has been controlled by the splash form and DDECalendar.

When any of the three combo boxes for the car type, seating or storage are selected by clicking or tabbing, the **Clear_Car_List** procedure in **mod_Car_Request** is executed. The purpose of this procedure is to reset the list of available cars whenever any of the requirements are modified.

```
Sub Clear_Car_List()
' Simply resets the car list anytime one of the request criteria
' has changed.
  With frm_Car_Request.cmb_Car
    .Clear
    .AddItem "None"
    .ListIndex = 0
  End With
End Sub
```

As explained earlier, the user must first build the list of available cars before selecting a car by clicking the Find available cars button. It would probably have been better to build this list in the DropDown event of the Cars combo box. When the button is clicked, the procedure for that event calls the **Get_Available_Cars** procedure which interacts with the Excel spreadsheet to generate the car list.

```
Sub cm3_Find_Cars_Click ()
   cmb_Car.Clear
   Call Get_Available_Cars(txt_Date.Text)
   cmb_Car.ListIndex = 0
End Sub
```

The procedure **Get_Available_Cars**involves the most complex DDE conversations of this application. It begins by establishing a link with the worksheet for the appropriate month. Several text box controls, hidden by locating them off of the edge of the form's view, are used for the DDE links. By looping through each of the columns for the requested date, we can determine which cars are available. Whenever the cell for a specific date (row) and vehicle (column) is empty then it's available.

```
Sub Get_Available_Cars(ByVal requested_date As String)
  Dim column_index As Integer
  Dim storage_class As Integer

' Check the availability of each car for the given day.
  For column_index = 2 To 12
    With frm_Car_Request.txt_Available
```

```
        .LinkMode = vbLinkNone
        .LinkTopic = "Excel|" & DATA_FILE & month_of_request
        .LinkItem = "R" & Trim(Str(date_of_request + 1)) & "C" &↵
           Trim(Str(column_index))
        .LinkMode = vbLinkAutomatic
        .LinkMode = vbLinkNone
      End With
```

When a car is found, its specifications are checked to determine if it meets the minimal seating and storage requirements. Seating and storage capacities are kept at the bottom of the column for each car on a worksheet.

```
  ' If the specified car is available check if it meets minimum seating
  ' and storage requirements. If so, add it to the list.
      If (Asc(Left(frm_Car_Request.txt_Available.Text, 1)) = 13) Then

  ' Get seating specs.
        With frm_Car_Request.txt_Seating
          .LinkMode = vbLinkNone
          .LinkTopic = "Excel|" & DATA_FILE & month_of_request
          .LinkItem = "R34" & "C" & Trim(Str(column_index))
          .LinkMode = vbLinkAutomatic
          .LinkMode = vbLinkNone
        End With

  ' If car meets the seating specs check its storage specs.
        If (Val(frm_Car_Request.txt_Seating) >= Val(frm_Car_Request.↵
           cmb_Seating.List(frm_Car_Request.cmb_Seating.ListIndex))) Then
          With frm_Car_Request.txt_Storage
            .LinkMode = vbLinkNone
            .LinkTopic = "Excel|" & DATA_FILE & month_of_request
            .LinkItem = "R35" & "C" & Trim(Str(column_index))
            .LinkMode = vbLinkAutomatic
            .LinkMode = vbLinkNone
          End With

          Select Case Left$(frm_Car_Request.txt_Storage,↵
                 Len(frm_Car_Request.txt_Storage) - 2)
            Case "Small"
              storage_class = 0
            Case "Medium"
              storage_class = 1
            Case "Large"
              storage_class = 2
          End Select
```

When a vehicle is found to be both available and also meets the minimum storage and seating requirements, it's added to the available car list. Before adding a name, it must be trimmed of a trailing tab delimitation. The real trick here is that we make use of the **ItemData** property of the list box to store the column index for each vehicle. If this feature wasn't utilized, we would have to search the list of cars by looping through the Excel spreadsheet using DDE, looking for a match every time we went to reserve a car.

```
  ' If the car meets the storage specs add it to the list.
          If (storage_class >= frm_Car_Request.cmb_Storage.ListIndex) Then
            With frm_Car_Request
              .txt_Car.LinkMode = vbLinkNone
```

```
            .txt_Car.LinkTopic = "Excel|" & DATA_FILE & month_of_request
            .txt_Car.LinkItem = "R1" & "C" & Trim(Str(column_index))
            .txt_Car.LinkMode = vbLinkAutomatic
            .cmb_Car.AddItem Left(frm_Car_Request.txt_Car,↵
               Len(frm_Car_Request.txt_Car) - 2)
            .cmb_Car.ItemData(frm_Car_Request.cmb_Car.NewIndex) =↵
               column_index
            .txt_Car.LinkMode = vbLinkNone
          End With
        End If
      End If
    End If
  Next column_index

End Sub
```

Clearly, the list of available cars will vary depending upon your criteria.

All that's left is to reserve a car. This is performed by clicking on the Take car command button. The click event procedure for this control reserves the selected vehicle. It begins by confirming that all data necessary for reserving a car is available.

```
Private Sub cm3_Take_Car_Click()
  Dim date_of_request As Integer

' Verify that a name has been entered.
  If (Len(txt_Name.Text) = 0) Then
    Beep
    txt_Name.SetFocus
    Exit Sub
  End If

' Verify that a car is available.
  If (cmb_Car.List(cmb_Car.ListIndex) = "None") Then
    Beep
    cmb_Car.SetFocus
    Exit Sub
  End If
```

Once this has been determined, the reservation is performed. The coordinates for the Excel cell to fill for the reservation are determined by using the month, date and **ItemData** property of the Cars combo box. A link is established with the cell and its value is updated using the **LinkPoke** method.

```
' Reserve the selected car.
  date_of_request = Val(Format$(txt_Date.Text, "dd"))
  With txt_Name
    .LinkMode = vbLinkNone
    .LinkTopic = "Excel|" & DATA_FILE & month_of_request
    .LinkItem = "R" & Trim(Str(date_of_request + 1)) & "C" &↵
         Trim(cmb_Car.ItemData(cmb_Car.ListIndex))
    .LinkMode = vbLinkManual
    .LinkPoke
  End With
```

Finally, using the **Save_Data** procedure, the spreadsheet is saved with our new reservation.

```
    ' Save the spreadsheet with the new addition.
      Save_Data

    End Sub
```

Save_Data utilizes the **LinkExecute** method to instruct Excel to save the workbook.

```
    Sub Save_Data()
    ' Issues command to save the car request spreadsheet.
      With frm_Car_Request.txt_Month
        .LinkMode = vbLinkNone
        .LinkTopic = "Excel|" & DATA_FILE_ONLY
        .LinkMode = vbLinkManual
        .LinkExecute "[Save]"
      End With
    End Sub
```

Summary

This simple Visual Basic to Excel DDE example shows the power and extendibility that is available through the utilization of other Windows applications with your programs.

You may be wondering why we've covered DDE when it's an aging technology being pushed aside by OLE as the standard means of data exchange and program control. There are three main reasons:

- Many applications support DDE, but not OLE.
- DDE is probably the simplest method for continuously updating data between applications.
- You can use DDE across a network in a way that you can't use OLE, unless you have the help of the Enterprise edition of Visual Basic.

Let's take a look now at how DDE handles network communication.

NetDDE

NetDDE, just like it sounds, is Dynamic Data Exchange over a network. Once set up it is essentially the same as the DDE we've already seen, but before you can use it there are a number of steps that need to be performed to get NetDDE working. You'll need to meet all of the following conditions before the example will work.

- A network adapter and appropriate networking software needs to be installed on at least two machines linked via a LAN. The network must include NetBEUI as one of its protocols.
- The source application must be registered in the HKEY_LOCAL_MACHINE/Software/Microsoft/NetDDE/DDE Shares section of the Registry in Windows 95, or in the [DDEShares] section of **System.ini** in Windows for Workgroups.
- The source application should be located somewhere in the server machine's path or an entry for it should be added to the HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\App Paths section of the Registry.
- **NetDDE.exe** must be started so that DDE communication can be transported over the network.

For our example **\Chap11\Chatter\Chatter.vbp**, you can find a sample Registry file in the project directory called **Chatter.reg**. You can merge this with your Registry by simply double-clicking it in Explorer.

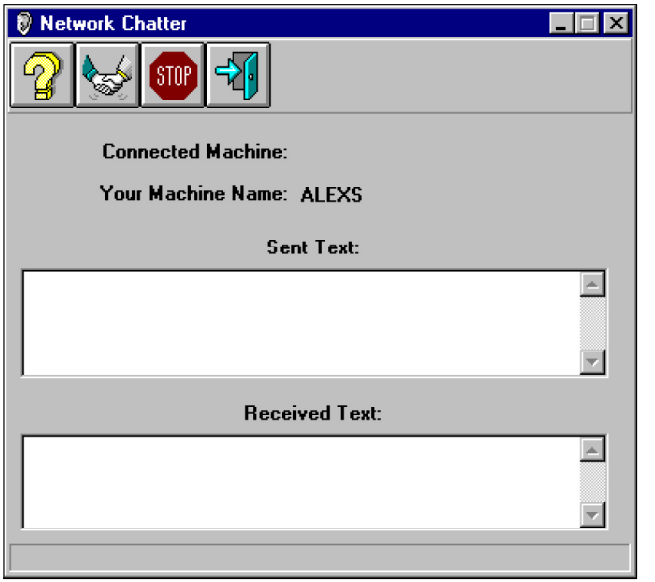
```
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NetDDE\DDE Shares\CHATTER]
"Additional item count"=hex:00,00,00,00
"Application"="chatter"
"Item"=""
"Password1"=""
"Password2"=""
"Permissions1"=hex:1f,00,00,00
"Permissions2"=hex:00,00,00,00
"Topic"="chatter"
```

*Note that **frm_Chatter** was set up as a DDE source with a **LinkTopic** of "Chatter".*

To register an application in the DDE Shares section of the Registry, you must provide at least three pieces of information: an alias in the form of a Registry key which you'll use to refer to the source application in code; and Registry values for both the application (executable) name and the link topic. In addition, you can also specify restrictions on the use of your server. For example, you can prevent users from using **LinkPoke** on your DDE server or specify that they must provide a password to access it. We'll just use the basic items in this application.

Network Chatter

Network Chatter shows how you can use NetDDE to create a communication application that allows users to converse with each other over a network. This is similar to the Chat application that comes with Windows. If you run the application, after making sure the requirements listed above have been met, you'll see the form shown below. The text boxes are blank apart from the one displaying the name of your computer. Use the handshake button to bring up a form that allows you to enter the name of the machine that you want to connect to.



Enter a machine name of another computer on the network that also has **Chatter.exe** installed then press OK. Chatter will attempt to attach to the specified computer.

If the computer doesn't have NetDDE running or if Chatter isn't in that computer's path, nothing much will happen, otherwise Chatter will be started on the machine you specified and a message will be passed to it requesting a conversation. If the user of that machine agrees to a conversation, they should click the handshake button and they will see the form shown above with the name of the calling machine already entered. Pressing the OK button will connect them to the caller so that a two-way conversation can ensue by typing into the SentText box and reading the messages in the ReceivedText box.

When either user has finished they can terminate the conversation either by closing the Chatter application down or by clicking the Stop button.

What's Happening?

The program begins with **Sub Main** located in **mod_Chatter**. This procedure ensures that NetDDE is running on our machine. Although we can't make sure that it's running on our target machine, we can at least make sure that it's running on our own.

```
Sub main()
Dim return_code As Long

  On Error Resume Next
  return_code = Shell("netdde.exe", vbHide)
  frm_Chatter.Show
  On Error GoTo 0
End Sub
```

The **Form_Load** procedure for **frm_Chatter** centers the form and retrieves the computer name using the **GetComputerName** API function. This API call is 32-bit only so if you need to use this code on 16-bit Windows, I'm afraid you'll have to find your own alternative.

```
Option Explicit
#If Win32 Then
  Private Declare Function GetComputerName Lib "kernel32" Alias↵
          "GetComputerNameA" (ByVal lpBuffer As String, nSize As Long)↵
          As Long
#End If

Function Get_Computer_Name() As String
Dim return_code As Long
Dim computer_name As String
Dim computer_name_length As Long

  computer_name_length = 255
  computer_name = String$(computer_name_length, "J")

' Get the current computer's name and return it.
  return_code = GetComputerName(computer_name, computer_name_length)
  Get_Computer_Name = Left$(computer_name, computer_name_length)

End Function
```

Once loaded, the application will sit there until the user presses a button or another machine attempts to connect to this one. We'll look through the connection process from the point of view of a user who's initiating the conversation.

The user starts to initiate a conversation by pressing the handshake button. This shows **frm_Get_Username**. Once the destination computer name has been entered, the user will press the OK button. The click procedure for this button adds the destination computer's name to the text box on the main form and calls that forms **Open_NetDDE** method to start the conversation.

```
Private Sub cmd_OK_Click()
' Place the destination computer name in the main form's computer name
' text box.
  frm_Chatter.txt_Dest_Comp_Name.Text = UCase$(txt_Dest_Comp_Name.Text)
  frm_Chatter.Open_NetDDE frm_Chatter.txt_Dest_Comp_Name.Text
  Unload Me
End Sub
```

The **Open_NetDDE** procedure starts by performing a few initializations to let the user know that a potentially lengthy process is about to occur.

```
Public Function Open_NetDDE(dest_computer_name As String) As Boolean
Dim retry_count As Integer

  If dest_computer_name = "" Then
    Exit Function
  End If

  On Error Resume Next
  Screen.MousePointer = vbHourglass
  DoEvents

  Open_NetDDE = False
  retry_count = 0
  Err = 0

  pn3_status.Caption = "Attempting to connect to computer: " &
         dest_computer_name
```

At this stage, the function enters a loop to try and get the other machine to respond. It uses the same principle that we've seen earlier: it attempts a link and checks for errors. If it detects an error, the link is canceled and it attempts to create a new one by looping back to the top. It will only make ten attempts to link successfully.

```
Do While (retry_count < 10)
  txt_Received.LinkTopic = "\\" + dest_computer_name + "\NDDE$|CHATTER"
  txt_Received.LinkItem = "txt_Sent"
  txt_Received.LinkMode = vbLinkAutomatic

    If Err <> 0 Then
    ' Cancel the connection, add 1 to the retry count and try again
      txt_Received.LinkMode = vbLinkNone
      retry_count = retry_count + 1
```

Note the format that we're using for the **LinkTopic** property. In general, you should use "*machine_name***NDDE$|***application_share_name*" as the topic for your NetDDE links where ***machine_name*** is the name of the machine that you would like to run a particular application, and ***application_share_name*** is the alias for that application declared in the NetDDE section of that machine's Registry.

If the link is opened successfully, then we have succeeded in linking our **txt_Received** text box to the other machine's **txt_Sent** text box. Since we have used an automatic link, whenever the other user types something into their upper text box, we will automatically receive it in our lower text box. At the moment, we could have a one-way conversation with us on the receiving end, but to establish a two way conversation we need to get the other user to link to us.

We do this by poking a text string into the other machine's **txt_Received** box requesting a conversation. This is accomplished by establishing a link between the **txt_Command_Output** text box on this machine and the **txt_Received** text box on the other, using **LinkPoke** to pass the string and then breaking the link.

```
      Else
      ' We have a connection so set Open_NetDDE to True
        Open_NetDDE = True
      ' Pass our computer name
        With txt_Command_Output
          .LinkTopic = "\\" + dest_computer_name + "\NDDE$|CHATTER"
          .LinkItem = "txt_Received"
          .LinkMode = vbLinkAutomatic
          .Text = "Chat with " & Get_Computer_Name() & "?"
          .LinkPoke
          .LinkMode = vbLinkNone
        End With
```

We then link the same text box to the **txt_Command_Input** box on the other machine and pass it a command string telling it that we'd like to initiate a conversation, and passing it our computer name.

```
      ' Pass the initiate message
        With txt_Command_Output
          .LinkTopic = "\\" + dest_computer_name + "\NDDE$|CHATTER"
          .LinkItem = "txt_Command_Input"
          .LinkMode = vbLinkAutomatic
          .Text = "Initiate" & Get_Computer_Name()
          .LinkPoke
          .LinkMode = vbLinkNone
        End With

        pn3_status.Caption = "DDE successful"
        Exit Do
      End If

    ' reset the vb error code to clear.
      Err = 0
    Loop

    Screen.MousePointer = vbDefault
    On Error GoTo 0

  End Function
```

The change event for the command input box on the other machine is triggered when we pass it our computer name. This procedure puts the name that we pass into the destination computer box on **frm_Get_Username** so that if the user agrees to the conversation, they won't need to type in the computer name themselves when they bring up that form by pressing the handshake button.

```
Private Sub txt_Command_Input_Change()
' Are we being asked to terminate the chat?
  If InStr(txt_Command_Input.Text, "Terminate") Then
  ' Are we the one asking for the termination
    If (Mid$(txt_Command_Input.Text, 10) = txt_Source_Comp_Name.Text) Then
      Exit Sub
    Else
      Disconnect_NetDDE
    End If
  End If

' Is a Chat being initiated?
  If InStr(txt_Command_Input.Text, "Initiate") Then
  ' Are we doing the initiating?
    If (Mid$(txt_Command_Input.Text, 9) = txt_Source_Comp_Name.Text) Then
      Exit Sub
    Else
      frm_Get_Username.txt_Dest_Comp_Name.Text =↵
         Mid$(txt_Command_Input.Text, 9)
    End If
  End If
End Sub
```

If the other user does agree to the conversation, they'll connect to us in exactly the same way as we connected to them: they'll link their receive text box so that it automatically receives the text from our send text box, and so a two-way link is created with Chatter acting both as a destination and as a source on each machine.

This conversation will continue until either of the parties decides to end it either by closing their application or clicking the Stop button. In either case, the **cm3_Disconnect_Click** procedure is executed.

```
Private Sub cm3_Disconnect_Click()
  Notify_Termination
  Disconnect_NetDDE
End Sub
```

The first procedure called sends some text from the **txt_Command_Output** text box of the closing machine to the **txt_Command_Input** box of the other, notifying it of the termination of the conversation. As you can see from the code in the **txt_Command_Input_Change** procedure shown above, this will cause the other application to execute the **Disconnect_NetDDE** procedure too. We need to make sure that this happens, so that the other machine doesn't continue to receive text from our send text box. The **Disconnect_NetDDE** procedure just makes sure that all possible links to the other machine's Chatter program are closed.

```
Private Function Disconnect_NetDDE() As Boolean
  If txt_Dest_Comp_Name.Text = "" Then
    Exit Function
  End If

  On Error Resume Next
  DoEvents
  pn3_status.Caption = "Attempting to disconnect from other machine."

  txt_Dest_Comp_Name.Text = ""

' Clear the sent text box.
```

```
    txt_Sent.Text = ""

    txt_Source_Comp_Name.Text = ""
    txt_Source_Comp_Name.LinkPoke

  ' cancel the NetDDE connection
    txt_Source_Comp_Name.LinkMode = vbLinkNone

  ' Disconnect the received text box.
    txt_Received.LinkMode = vbLinkNone
    txt_Received.Text = ""

    txt_Command_Input.LinkMode = vbLinkNone
    txt_Command_Output.LinkMode = vbLinkNone

    Disconnect_NetDDE = False
    On Error GoTo 0
    txt_Source_Comp_Name.Text = Get_Computer_Name()
    pn3_status.Caption = "Disconnected from other machine."


  End Function
```

Summary

You can see that DDE is a very useful tool, despite the rise of OLE. The fact that NetDDE allows you to execute and communicate with applications running on other machines makes it more useful still. Although our application is a fairly trivial one, you can use these NetDDE techniques to create your own network applications with very little effort.

Communicating with MAPI

MAPI is the Messaging Application Programming Interface, an API that lets you take advantage of the facilities of any MAPI-compliant mail systems on your users' systems. By using the MAPI functions or MAPI controls with Visual Basic, you can write code to integrate your own applications with your users' existing mail system, allowing you to perform all sorts of mail manipulation tasks from your own code.

In this next section, we're going to mail-enable the MicroPost application that we created earlier in the book. We'll do this using the MAPI Session and MAPIMessages controls supplied with the Professional edition of Visual Basic 4. Before we do this, you'll need a MAPI mail system on your machine to test it. If you don't already have a MAPI mail system on your machine, you can install one by installing Microsoft Mail that comes with Windows NT, Windows for Workgroups or Windows 95. On Windows 95, you'll have to install Exchange to get Microsoft Mail. Once your mail system is installed and configured you can test the new version of MicroPost, even if you can only send mail to yourself.

MicroPost Mail

You'll find MicroPost Mail in the **\Chap11\MPstMail** subdirectory on the disc. If you run the project **Mpstmail.vbp**, you'll see that relatively little has changed since the first version; all of the original features remain. In addition, the main form has gained three menu items: Connect to E-mail, Get a Note from E-mail, and Disconnect from E-mail while each of the notes has gained a single menu item: Mail.

Before you can use any of the mail facilities that MicroPost Mail offers, you need to connect to your e-mail system by clicking the Connect to E-mail item in the Notes menu of the main form. This will bring up the standard login screen for your mail system, allowing you to choose a mail profile or enter a password as necessary.

Once connected to your mail system, the other menu items that were previously disabled become enabled so that you can perform mail operations on your notes. Hitting the Mail item on one of your notes allows you to send it to anyone you choose, and clicking on the Get a Note from E-mail item on the main form lets you choose one of the messages in your in-box to turn into a MicroPost note. When you've finished with the mail system, you can disconnect by hitting the Disconnect from E-mail menu item or just close the application.

What's Happening?

We'll begin our look behind the scenes by seeing how we connect to the mail system. The click event for this menu item just calls the procedure **Connect_to_Mail_System** located in **frm_Micropost**.

```
Private Sub mnu_Notes_Connect_Mail_Server_Click()
  Connect_to_Mail_System
End Sub
```

Connecting to MAPI

Connecting to a MAPI mail system is as easy as using the **SignOn** method of the MAPISession control, **mps_Login**. If it fails to connect to the user's mail system, an error will be generated which we can trap and handle appropriately. Otherwise, we need to extract the **SessionID** from the MAPISession control and pass it to the MAPIMessages control, **mpm_Micropost**. This ensures that both controls are working with the same mail session.

```
Sub Connect_to_Mail_System()
Dim response As Integer

  On Error Resume Next
  Err = 0
' Attempt to connect to a mail server
  mps_Login.SignOn
  If Err = 0 Then
    mpm_Micropost.SessionID = mps_Login.SessionID
    mail_enabled = True
  Else
    response = MsgBox("Error when attempting to make" & Chr(vbKeyReturn)↵
         & Chr(10) & "a connection to E-Mail system.", vbOKOnly +↵
         vbDefaultButton1 + vbInformation + vbApplicationModal, "Mail↵
         Login")
    mail_enabled = False
  End If
```

Once we've determined whether we've opened a mail session, then we can call the **Enable_Mail_Menus** procedure to enable and disable the menu items relating to mail depending on our success.

```
  Enable_Mail_Menus mail_enabled
  On Error GoTo 0
End Sub
```

Here, once again, you can see how useful the **visible_notes** collection has been.

```
Sub Enable_Mail_Menus(ByVal enabled_status As Boolean)
  Dim note_object As frm_Note

' Enable/disable main form mail menu items.
  mnu_Notes_Connect_Mail_Server.Enabled = Not (enabled_status)
  mnu_Notes_Disconnect_Mail_Server.Enabled = enabled_status
  mnu_Notes_Get_Mail_Note.Enabled = enabled_status

' Enable/disable mail menu item on all Visible Notes.
  For Each note_object In visible_notes
    If enabled_status Then
      note_object.EnableMail
    Else
      note_object.DisableMail
    End If
  Next note_object

End Sub
```

Signing Off

The procedure to disconnect from the mail system is even simpler as it only uses the **SignOff** method of the **mps_Login** MAPISession control. This procedure is called from the menu item click event and the **Unload** procedure of the main form.

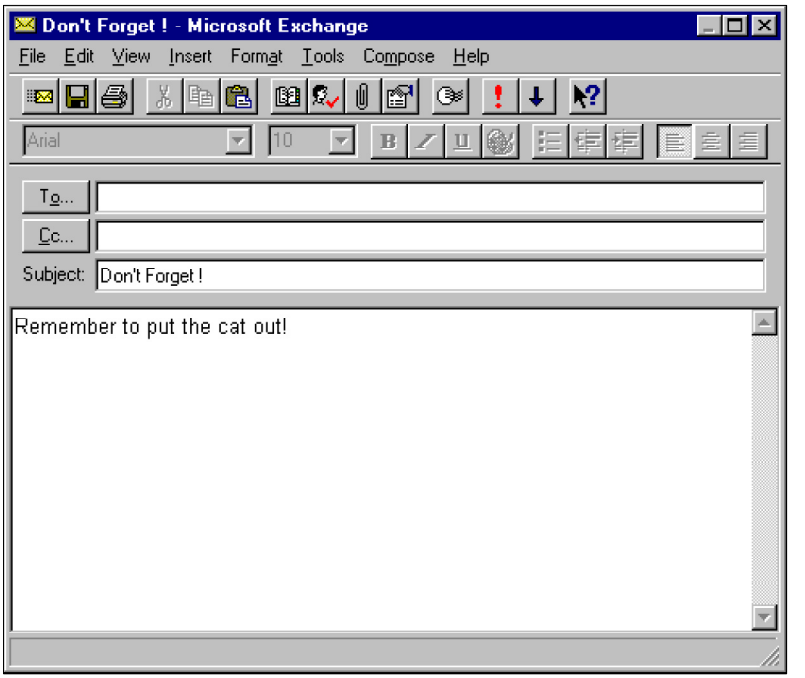
```
Sub Disconnect_from_Mail_System()
  On Error Resume Next
' Disconnect from Mail Server
  mps_Login.SignOff

' Disable/Enable Mail menu items.
  mail_enabled = False
  Enable_Mail_Menus mail_enabled
  On Error GoTo 0
End Sub
```

Mailing the Notes

The mailing of individual notes is achieved through the new **MailMe** method provided for them. Although we only call it from the note that's to be sent, the fact that it's a **Public** method means that it offers additional flexibility. For example, it would only take a few lines of code to send all the notes:

```
Public Sub MailMe()
' Mail current note.
  With frm_MicroPost.mpm_Micropost
    .Compose
    .MsgNoteText = txt_Note.Text
    .MsgSubject = Me.Caption
    .Send True
  End With
End Sub
```

The **Compose** method of the MAPIMessages control indicates that we're composing a new message. We set the text of the message to be the text of the note that is to be sent and set its subject from the **Caption** of the note by configuring the **MsgNoteText** and **MsgSubject** properties of the control. The **Send** method sends the note, with the parameter determining whether the send message dialog should be displayed. Since we need to find out who the message should go to, and we don't intend to create our own dialog in order to get this information, we use **True** as the parameter so that the dialog is displayed.



Retrieving a Message

Retrieving a message and creating a note from it is just as easy. This requires us to add a new form to the project, **frm_Get_Mail_Note**, from which the user can choose a message to make into a note. The **Form_Load** procedure for this form just centers the form and calls the **Load_Mail** procedure also located within this form.

```
Private Sub Form_Load()
  Center_Form Me
  Load_Mail
End Sub
```

Load_Mail begins by configuring the **mpm_Micropost** control to retrieve the messages in the order they were received by setting the **FetchSorted** property to **True**. Setting this property to **False** would retrieve the messages in the sort order specified by the user's Inbox. You can also use the **FetchMsgType** and **FetchOnlyUnread** properties to further define the messages to be fetched.

The **Fetch** method actually retrieves the messages and populates the control's message set.

```
Sub Load_Mail()
Dim count As Integer

' Get all the Notes from the selected mail system and place them in the
' list box.
  On Error Resume Next
  Err = 0
  frm_MicroPost.mpm_Micropost.FetchSorted = True
  frm_MicroPost.mpm_Micropost.Fetch
  lst_Mail_Notes.Clear
```

Displaying Messages

The number of messages can be read from the **MsgCount** property of the MAPIMessages control, and information can be extracted from a message set by setting the control's **MsgIndex** property to the desired number and reading any of a number of properties relating to the message.

In the code below, you can see that we're using the **MsgIndex** property to loop through all of the messages in the message set, and add the sender name (**MsgOrigDisplayName**) and the subject (**MsgSubject**) of each message to the list from which the user can pick a message to display. We store the **MsgIndex** as the **ItemData** property of each list item so that *we* can easily retrieve whichever message the user chooses to display.

```
    For count = 0 To frm_MicroPost.mpm_Micropost.MsgCount - 1
      frm_MicroPost.mpm_Micropost.MsgIndex = count
      lst_Mail_Notes.AddItem frm_MicroPost.mpm_Micropost.MsgOrigDisplayName↵
           & "    " & vbTab & frm_MicroPost.mpm_Micropost.MsgSubject
      lst_Mail_Notes.ItemData(lst_Mail_Notes.NewIndex) = count
    Next count
    On Error GoTo 0

  End Sub
```

When the user chooses a message, we create a new note for it in the **cm3_Okay_Click** procedure as shown below. This is essentially the same as the **Create_New_Note** procedure found in **mod_Micropost**, except that we use the **MsgSubject** of the chosen note to configure the title and the **MsgNoteText** property for the text of the note.

```
  Private Sub cm3_Okay_Click()
  ' Add a New Note based on the selected mail message.
    frm_MicroPost.mpm_Micropost.MsgIndex =↵
           lst_Mail_Notes.ItemData(lst_Mail_Notes.ListIndex)
    With note
      .note_ID = 0
      .Title = frm_MicroPost.mpm_Micropost.MsgSubject
      .Top = defaults_SET("Top")
      .Left = defaults_SET("Left")
      .Height = defaults_SET("Height")
      .Width = defaults_SET("Width")
      .FontName = defaults_SET("Font_Name")
      .FontSize = defaults_SET("Font_Size")
      .FontBold = defaults_SET("Font_Bold")
      .FontItalic = defaults_SET("Font_Italic")
      .FontUnderline = defaults_SET("Font_Underline")
      .FontStrikeThrough = defaults_SET("Font_StrikeThrough")
      .ForeColor = defaults_SET("ForeColor")
      .BackColor = defaults_SET("BackColor")
      .Memo = frm_MicroPost.mpm_Micropost.MsgNoteText
    End With

  ' Store the note.
    Save_Note

  ' Display the note.
    Call Add_A_Note

    Unload Me
  End Sub
```

As you can see from this example, mail-enabling your applications is a simple task with the MAPISession and MAPIMessages controls. These controls are more flexible than we have shown here, allowing you to build more of the mail functions into your own user interface, but most of the time you'll probably want to use the user's existing mail interface, as that is what they'll feel comfortable with.

If you're worried about the slight overhead incurred by using an OCX rather than raw code, then instead of the MAPI controls, you could call the MAPI functions directly just as you would any other API functions. You can find documentation for these functions in the MSDN/VB Starter Kit under Product Documentation\SDKs\Simple MAPI SDK. You won't find the declarations for the MAPI functions in the **Win32api.txt** file, but if you really need to use these functions directly, the documentation in the Starter Kit does contain the declarations in Visual Basic format.

Summary

This chapter has shown several different methods of extending your applications through the use of Visual Basic's communication capabilities. While, typically, we think of the serial port when discussing communication programs, in this chapter you were exposed to a variety of ways to add functionality to your programs through the use of communication techniques. Along the way you were shown how to:

- Work with your PC's serial port using the MSComm control.
- Develop Visual Basic DDE destination and server applications.
- Use NetDDE to work with other Windows programs across network.
- Extend your applications with the power of MAPI.