Visual Basic 5
Unsupported Controls and Utilities

Updated March 2, 2003

Dave Liske
Delmar Computing Services
http://www.mvps.org/htmlhelpcenter

When you first load a Visual Basic 5 CD into your machine and run **`setup.exe`**, you're presented with the Master Setup dialog box, which acts more like a web page and duplicates the artwork from the Visual Basic packaging. The third hyperlink from the top is labeled "Explore the CD". If you're like most of us, you went right past this and clicked on "Install Visual Basic" so you could get into it! However, exploring the CD is something we should go back to. Some of the Microsoft development system CD's contain treasures most people are unaware of, and this one is no exception.

We're going to go back to the CD now and explore it a bit more. The **`\tools\unsupprt`** directory contains a large number of controls and utilities we know absolutely nothing about. A slew of these are also useful in developing GUI's for our own applications.

It almost appears as if these controls and utilities were Microsoft in-house samples they decided were good enough to give out. I've also heard a rumor that runs along these lines: "When the programmers are coding the extras (bell and whistles) of the program, sometimes the program manager will cut them off and say 'No more, I don't care if it's not done, they aren't necessary to begin with and we have to meet the ship date'". Regardless of why they're there, or where they came from, the reality is that some of them still need a little work, but it's nothing which can't be overcome.

In this chapter we'll be looking at:

- The MSVBCalendar Control
- Dialog Automation Objects
- The System Tray Icon Control

I've included the these items since they're freely distributable, along with the sample code we'll be using throughout this chapter. Also, if you're using Visual Basic 6, you'll find the Dialog Automation Objects are included in the files for the Package And Deployment Wizard as dlgobjs.dll. This file has the same version number as the one we're using.

The Disclaimer

Yes, we need to have one of these! Workarounds and "fixes" are going to seem commonplace with some of these items. But at the same time, we've got the code in most cases. Sooner or later we'll figure out why these things don't work quite right and we'll be able to fix them. For now, however, we may have to kludge some of it the best we can.

These, and the other items in the **`\tools\unsupprt`** directory of the Visual Basic 5 CD, are **NOT** supported by Microsoft. This is better explained by the following sentence from the

bottom of the Help | About box for the MSVBCalendar control, which we'll be looking at shortly:

"Warning: This is an unsuported (sic) **sample control. Microsoft takes no responsibility for this control and no official support is available for this control."**

Even though I'm is providing a bit of documentation for some of these items through this chapter, I can't take responsibility for them either. Remember, if you intend to use and/or distribute these items in your own applications, test it, retest it, then test it again. When you're done with that, TEST IT SOME MORE! You'll be the one getting the telephone calls if you don't! And then, who ya' gonna' call?!?

But these goodies are worth more than just a brief look. They're quite nicely done and they deserve some use. In some instances, they're also a new, rather "sideways" look at how to accomplish certain tasks which have been accomplished differently up until now.

We're going to start with something which has been a long time in coming …

The MSVB Calendar Control

Microsoft Access has always had a nice calendar control. Ever since the first calendar control was released in an earlier version of the Access Developer's Toolkit, some Visual Basic developers have been a little jealous. Sure, calendar controls for Visual Basic from third-party developers have come and gone, but Visual Basic has never been released with one of its own.

Well, it has come to pass! There's quite a nice calendar control in the `\tools\unsupprt` directory of the Visual Basic 5 CD that has some extensive capabilities. It's also supplied with a rather in-depth test program in order to show off the capabilities of, and enumerate the properties of, the calendar itself.

Features

An interesting aspect of the MSVBCalendar control is that it's been supplied not as a finished *.ocx, but in the form of complete Visual Basic 5 source code. This is something the Access developers have never had to their advantage. At the same time, this calendar is fully compatible with 32-bit versions of Access.

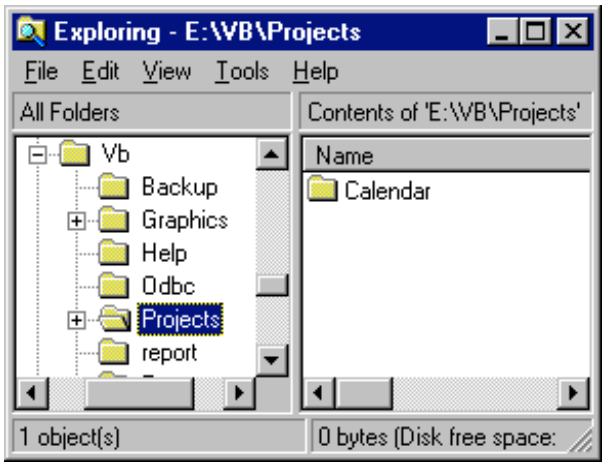
Also, the Access calendar control only handles the years from 1900 to 2100. In comparison, the Visual Basic Calendar Control is not only fully Y2K compatible, it uses a four-digit year field, and handles the years from 100 AD to 9999 AD without any problems. Since the source code is provided, the years could easily be extended even from this point outward. However, I don't know of a reason to do such a thing as of yet. Maybe if someone was writing an application for some kind of extensive history exam …

Getting Started

If you're copying the files from the Visual Basic 5 CD, we need to get the code for the control from the Visual Basic 5 CD and onto the local hard drive. I have a little place where I've placed these kinds of items so I can work with them more easily. We'll get the code onto the hard drive, change some file attributes, then get going with a look at the control itself.

Copying the Source Code

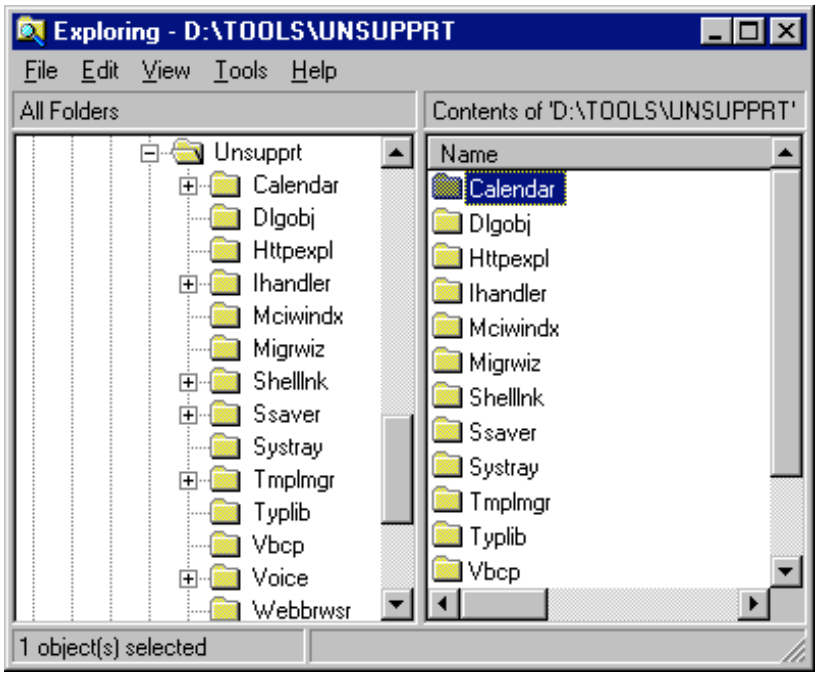
Create a place on your hard drive to store the source code and other files for the items we're going to be copying from the CD. For instance, the copy of Visual Basic 5 that I use is located in the **`e:\vb`** directory. Within this directory I've created a new folder and labeled it **`Projects`**:



*** PrjDir.bmp ***

This is the same type of directory used in other development packages for this purpose. In fact, Microsoft Visual C++ installs a directory similar to this during its own setup program and saves new projects to it by default.

Load the Visual Basic 5 CD into your CD drive and go to the **`c:\tools\unsupprt`** directory. Within this directory, highlight the **`Calendar`** directory:



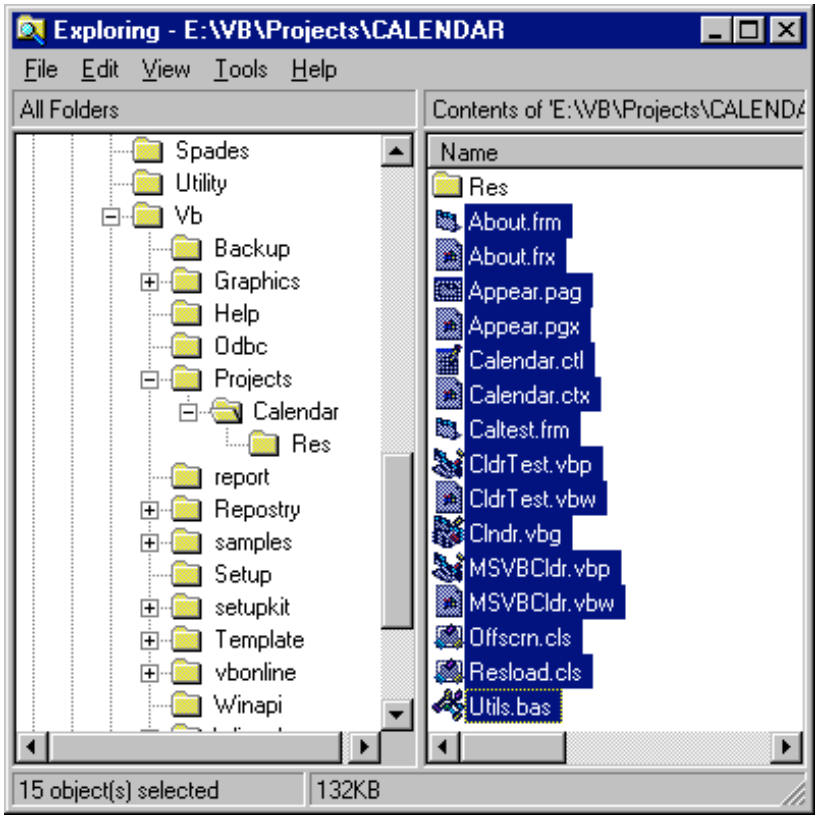
*** CalOrig.bmp ***

Right-click on this directory and click Copy from the pop-up menu.

Now, go back to your new **`Project`** directory, right-click on it and click Paste. This places all the source code for the Calendar Control where we can get to it.

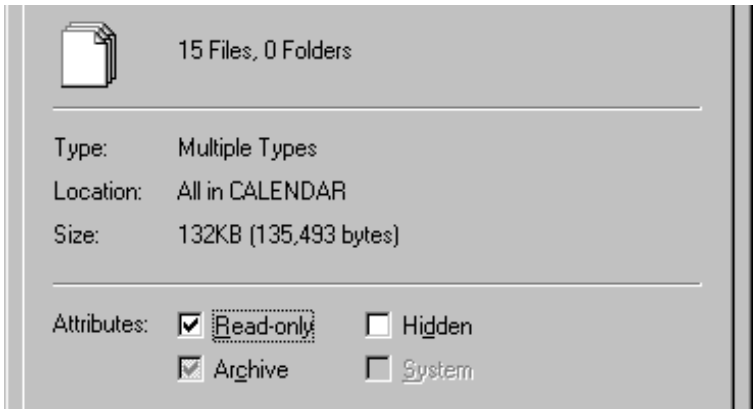
When we get ready to run the test program for the control, we'll do so before creating the control itself. At this point, the Visual Basic IDE will attempt to write information back to these files. Before this can occur, we need to change the attributes for these files since they are currently all "Read-only".

Open the directory on your hard drive containing the source code you just copied. Highlight all of the files, excluding the **`Res`** subdirectory. This is most easily accomplished by highlighting the first file in the directory, holding down the *Shift* key, then clicking on the last file in the directory.



*** H_Light.bmp ***

With these files highlighted, right-click on the whole list. Select Properties from the pop-up menu and the dialog box will appear:



*** PropDlg.bmp ***

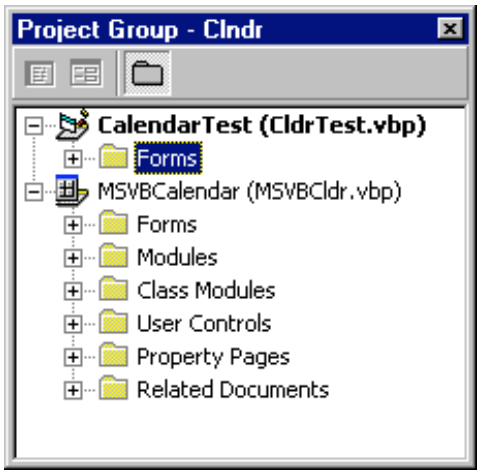
Clear the "Read-only" check box and click the OK button. Then, repeat the last part of this procedure with the contents of the **Res** subdirectory.

The Projects

Unless you've built a custom control with Visual Basic before, you may not yet be aware that Visual Basic is capable of hosting more than one project at a time in its IDE. The code for the calendar control is stored as a pair of projects, one for the control itself and one for its test program. These projects are named MSVBCldr.vbp and CldrTest.vbp respectively. There's also a third project within the files for this control, which is named Clndr.vbg. This is the group project, which gives the Visual Basic IDE the information necessary to open both of the other projects at the same time.

Looking At The Code

Start Visual Basic and open the project group Clndr.vbg. Once it loads, the Project Explorer will look something like this:

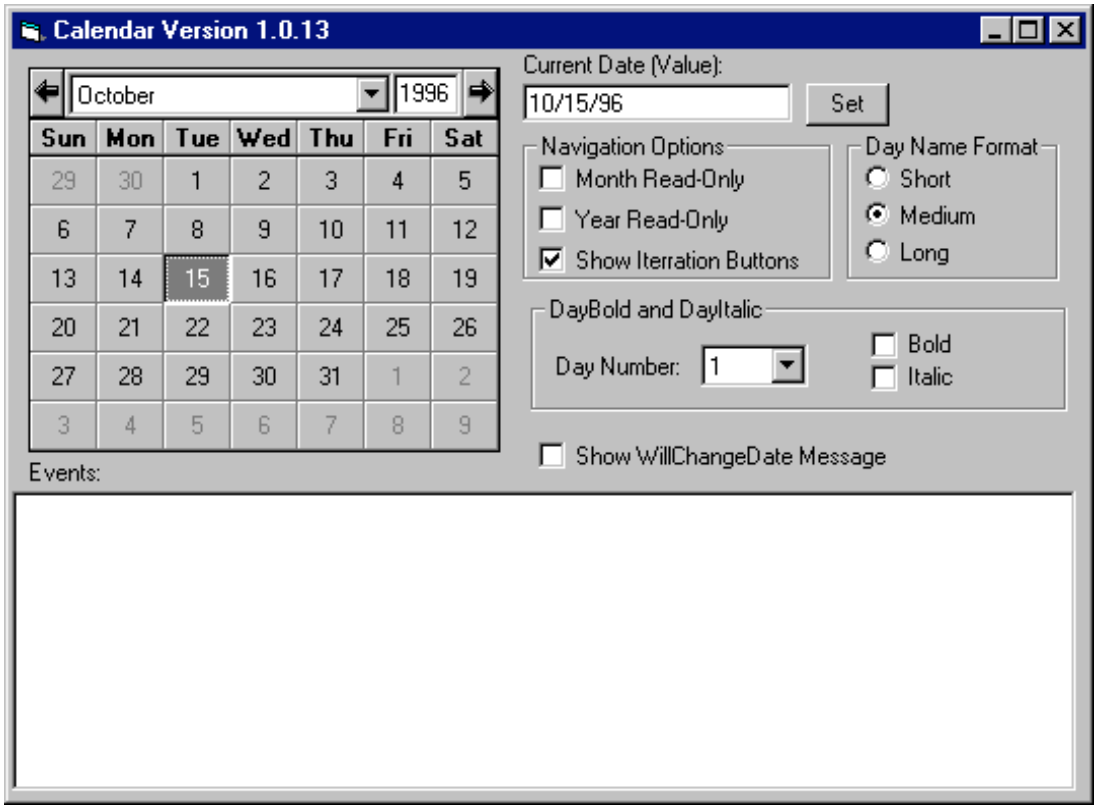


*** PrjGrp.bmp ***

Notice the calendar control MSVBCalendar is made up of at least one of everything, while the test project contains only a single form. Open this form in design mode, and you'll see that it is anything but simple. We're going to use this form in our discussion on the various features of the Calendar Control.

Try It Out - Running the Test Project

Press the *F5* key to start the test project. The CalendarTest project will be assembled, and will subsequently present its single form to you with the calendar in place.



*** CldrTest.bmp ***

> *Since CalendarTest is bolded in the Project Explorer, this is the project that will run when you either press the F5 key or select Run \ Start from the menubar of the Visual Basic IDE. This is set in the Project Explorer window. Place the mouse cursor directly over the CalendarTest heading and right-click. You'll see the top item on the popup menu is "Set as Start Up". If you do the same thing with the MSVBCalendar heading, you'll find this menu item is disabled. That's because a control project cannot be opened without a project to open it into. If there were another project in this group, one would have its "Set as Start Up" menu item checked and the other would not.*

Click on some of the items on the Test form. You'll notice that when you click on the calendar itself to change the highlighted day, or click on the command button with the caption 'Set', entries appear in the list box. Also, when you click on other items on the form, various properties of the control are changed.

Properties of the Calendar Control

The properties for the MSVBCalendar control are almost identical to those of the Access calendar control. So, if you've used the other control, this one will seem familiar. If you're not familiar with it, it's really not difficult to understand. Let's take a look at the properties:

Property	Description
Day	Returns/Sets the day number of the selected date.
DayColor	Returns/Sets the color used for the day numbers
DayFont	Returns/Sets the font used for the day numbers
DayNameColor	Returns/Sets the color used for the day names (i.e. days of the week)
DayNameFont	Returns/Sets the font used for the day names
DayNameFormat	Returns/Sets the format to use for the day names (calShortName = "M", calMediumName = "Mon", calLongName = "Monday")
Month	Returns/Sets the month number of the currently selected date (i.e. calJanuary = 1)
MonthReadOnly	Returns/Sets the read-only state of the month navigation combo box
ShowIterrationButtons	Returns/Sets the visible state of the previous and next month navigation buttons
StartOfWeek	Returns/Sets the first day of the week which will be displayed in the left-most column (calUse System = 0, calUseSunday = 1, … calUseSaturday = 7)
Year	Returns/Sets the year number of the currently selected date (4-digit integer)
YearReadOnly	Returns/Sets the read-only state of the year navigation text box
DataBindings	Returns/Sets a DataBindings Collection object that collects the bindable properties that are available to the developer
DataField	Returns/Sets a value that binds a control to a field in the current record
DataSource	Sets a value that specifies the data control through which the current control is bound to a database

The DataBindings, DataField, and DataSource properties aren't implemented in the form within the CalendarTest project since there's no database to go with this sample. If we need to connect this control to a database, we also need to provide a Data control on any form we design with this control in order to link the control to the correct database via the calendar's DataSource property.

Events of the Calendar Control

Two events can be seen listed in the list box in the test sample whenever you click on a date on the calendar or click on the 'Set' button. The first one is WillChangeDate, whose syntax looks like this:

```
WillChangeDate(ByVal NewDate As Date, Cancel As Boolean)
```

This event indicates that the currently selected date is about to change to the value NewDate. Immediately after this event, you'll see the listing for the occurrence of the DateChanged event:

```
DateChange(ByVal OldDate As Date, ByVal NewDate As Date)
```

This occurs immediately after the date value has been changed, and shows both the old and new dates.

Problem # 1 – Color Properties

DayColor and DayNameColor Properties

If you've played around with this control for a little while, you may have noticed something a bit peculiar regarding the DayColor and DayNameColor properties. Open the Test Project in Visual Basic's IDE and, while looking at the properties for the calendar control, change the DayColor property. You'll notice it will change for a brief moment, then changes to be the same as the DayNameColor property when the calendar gets refreshed. And if you change DayNameColor, the DayColor property also gets changed!

This problem is solved through changing the source code, as there's a bug in the code for the Get DayNameColor property. Let's have a look at this code:

```
'-----------------------------------------------------------------------
' DayNameColor Get/Let
'-----------------------------------------------------------------------
' Purpose:  Gets and sets the color used for the day numbers
'-----------------------------------------------------------------------
Public Property Get DayNameColor() As OLE_COLOR
    DayColor = mclrDayNames
End Property 'Get DayNameColor()
```

A simple mistake: setting the DayColor variable to mclrDayNames, instead of setting the Day**Name**Color variable to mclrDayNames. Forgetting part of a variable name is easy to do when the names are so similar. A possibility is that this code was developed from a copy of the code for the DayColor property, and that this was missed. This is something to remember when reviewing your own code.

What's Happening?

The Refresh property does exactly as its name suggests. One of the keys to its overall performance is that it carries out all of the **`Get`** properties in order to fully refresh a given control. When you set the DayColor property, the code for Let DayColor calls the Refresh routine:

```
Public Property Let DayColor(NewVal As OLE_COLOR)
    mclrDay = NewVal
    UserControl.Refresh
End Property 'Let DayColor()
```

When this occurs, one of the routines called during the Refresh is Get DayColor:

```
'-----------------------------------------------------------------------
' DayColor Get/Let
'-----------------------------------------------------------------------
' Purpose:  Gets and sets the color used for the day numbers
'-----------------------------------------------------------------------
Public Property Get DayColor() As OLE_COLOR
    DayColor = mclrDay
End Property 'Get DayColor()
```

This provides us with the grief glimpse of the correct color for the days. However, since the code for Get DayNameColor comes later in the list of routines than the Get DayNameColor property, the DayColor property ultimately gets set to the color for DayNameColor, while the DayNameColor Property doesn't get refreshed at all.

Almost the same thing occurs when you set the DayNameColor property, with a couple of minor exceptions. The instant the color is changed, mclrDayNames gets properly set. When the Refresh occurs for this routine, the DayColor gets set to DayNameColor as

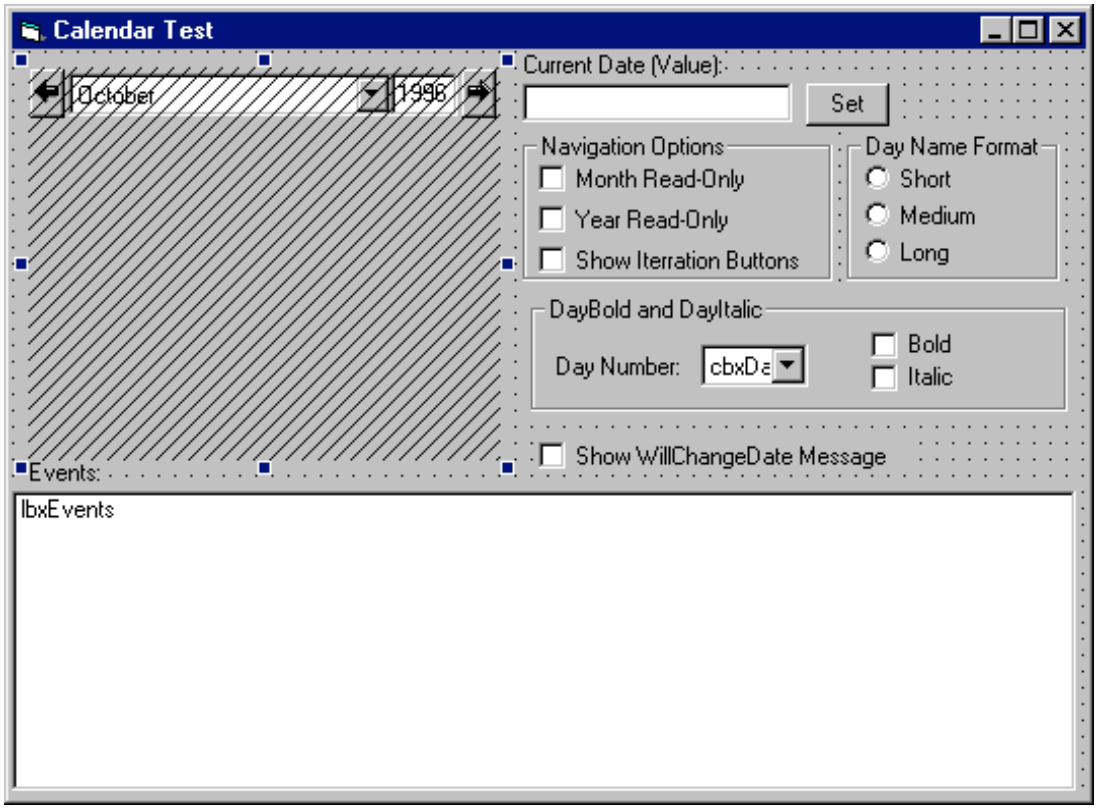
well, again via the Get DayNameColor property. Since the DayNameColor property is not addressed correctly in its Get routine, the box on the OLE_COLOR control in the properties menu for DayNameColor never changes.

The "Fix"

Repair the Get DayNameColor code as follows:

```
'------------------------------------------------------------------------
' DayNameColor Get/Let
'------------------------------------------------------------------------
' Purpose:  Gets and sets the color used for the day numbers
'------------------------------------------------------------------------
Public Property Get DayNameColor() As OLE_COLOR
    ' DayColor = mclrDayNames
    DayNameColor = mclrDayNames
End Property 'Get DayNameColor()
```

If you now try to view the test form in design mode without going any further, you'll see this:



*** ClrErr.bmp ***

The calendar control needs to be compiled once before we can check the results of our fix. Press the *F5* key to run the sample, then shut it down. The test form will now show up as it did the first time we looked at it. Now, go ahead and toy with the DayColor and DayNameColor properties, and you'll see that these properties now work correctly.

Once you've done this, select 'Make MSVBCldr.ocx' from the File menu of the IDE, making sure to select the **c:\windows\system** directory as its location.

Now What?

This problem was labeled #1, which means that, yes, there's another one we'll address a little later on. Does this indicate that we shouldn't use this control in our applications? Of course not. It's supplied as source code, so anything we find we should go ahead and repair. Problem #2 is a bit different, and we'll see how to repair something outside of the source code.

The reason we receive samples from Microsoft with our development software is to see how to do things, not only how to accomplish a particular task, but also how to develop software in a manner consistent with certain software industry standards.

These unsupported controls and utilities are a bit different. Looking deeper at some of the code for this particular control reveals that we are looking at a work in-progress that we normally wouldn't be allowed to see. For instance, the following code is from the Declarations section of the UserControl for the MSVBCalendar Control:

```
'------------------------------------------------------------------------
' Calendar.ctl
'------------------------------------------------------------------------
' Implementation file for the VB Calendar control sample.
' This control displays a month-at-a-time view calendar that the
' developer can use to let users view and adjust date values
'------------------------------------------------------------------------
' Copyright (c) 1996, Microsoft Corporation
'              All Rights Reserved
'
' Information Contained Herin is Proprietary and Confidential
'------------------------------------------------------------------------
```

The bolded line is important, and indicates that this control may not have been intended as a sample. This may have actually been a control intended to be shipped with Visual Basic 5 that was turned into an unsupported sample due to time constraints.

The code for the UserControl's Paint event is also interesting, as it reveals the uncompleted aspect of this sample:

```
'change the text color to dark gray to paint the previous month days
'daveste -- 7/31/96
'TODO: this should be replaced with day styles or at least with
 'a property the control the font and color of these other dates
 dcWork.TextColor = RGB(128, 128, 128)
```

Note the grammar on that one. Definitely incomplete!

There are other TODO lines throughout the code. You'll find them if you look around a bit more.

What's even more interesting is the following code, also from the UserControl portion of the code:

```
Private Sub CopyFont(fntSource As StdFont, fntDest As StdFont)
    'daveste -- 8/14/96
    'REVIEW:  Is there a better way to do this???!!!

    'if the destination is nothing, create a new font object
    If fntDest Is Nothing Then Set fntDest = New StdFont

    fntDest.Bold = fntSource.Bold
    fntDest.Charset = fntSource.Charset
    fntDest.Italic = fntSource.Italic
    fntDest.Name = fntSource.Name
    fntDest.Size = fntSource.Size
```

```
        fntDest.Strikethrough = fntSource.Strikethrough
        fntDest.Underline = fntSource.Underline
        fntDest.Weight = fntSource.Weight
    End Sub 'CopyFont()
```

Good question from the anonymous ("daveste"?) code reviewer, and quite a challenge for those of us who now have the code. The challenge? Quite simply, this: Finish designing the control. Complete the TODO items, answer the reviewers questions, and test the final control for problems like the one we just repaired. Customize it as you'd like, since you've got the source code and the ability to do so. The end result in all of this will not just be a nifty little calendar control that works to our liking. Rather, this is a great way to become better developers than we already are.

Try It Out – Usage of the MSVBCalendar Control

In the code provided with this chapter is a sample application we'll use to see how the calendar control can be used. Open the project chapter6.vbp and have a look at the form:

*** CalProj.bmp ***

If an error occurs during the loading of this project, it's likely due to the copy of MSVBCIdr.ocx on your system being not quite what the IDE is looking for. If this happens, the place where the calendar would have been shown will only show an empty 3-D rectangle. Delete this rectangle, and ensure that the Microsoft Visual Basic Calendar is selected in the projects' components. Then, add the calendar to the project, on this form, in the place the 3-D rectangle was previously. The default name for the calendar is Calendar1, which is what I used to develop this sample. None of the calendar's properties need to be changed.

Right now there is no code on the form that performs any kind of tasks. The menus are empty, as are the events of the calendar control itself.

Place the following code in the form's Load event to set the calendar's date to the current date each time the form is opened:

```
Private Sub Form_Load()

  ' Set the calendar to today's date
  With Calendar1
    .Year = Format(Now, "yyyy")
    .Month = Format(Now, "m")
    .Day = Format(Now, "d")
    .Refresh
  End With

End Sub
```

Place the following code under the DblClick event of the calendar control Calendar1:

```
Private Sub Calendar1_DblClick()


  ' Declare the variables
  Dim SelDate As Variant, NowDate As Variant
  Dim SelYear As Integer, SelMonth As Integer, SelDay As Integer

  ' Assemble the date string from the calendar control
  With Calendar1
    SelYear = .Year
    SelMonth = .Month
    SelDay = .Day
  End With
  SelDate = DateSerial(SelYear, SelMonth, SelDay)

  ' Compute today's date
  NowDate = DateSerial((Format(Now, "yyyy")), (Format(Now, "m")), (Format(Now, "d")))

  ' Test to see if the selected date is earlier than today
  If SelDate < NowDate Then

    'If it is...
    Msg = "The  date you have selected is earlier than today." & Chr(10) & _
      "Please select today's date or later."
    Style = vbOKOnly + vbExclamation
    Title = "Date Selection Error"
    Response = MsgBox(Msg, Style, Title)

    ' Reset the calendar
    With Calendar1
      .Year = Format(Now, "yyyy")
      .Month = Format(Now, "m")
      .Day = Format(Now, "d")
      .Refresh
    End With

  Else
    ' Load the text box with the selected date
    If Option1.Value = True Then
      ' U.S.A.
      Text1.Text = SelMonth & "/" & SelDay & "/" & SelYear
    Else
      If Option2.Value = True Then
        ' Europe
        Text1.Text = SelDay & "/" & SelMonth & "/" & SelYear
        Else
          ' U.S.A.
          Option1.Value = True
          Text1.Text = SelMonth & "/" & SelDay & "/" & SelYear
      End If
    End If
  End If
```

```
End Sub
```

Place the following code under the option button labeled U.S.A.:

```
Private Sub Option1_Click()

  ' Load the text box with the selected date
  If Text1.Text = "" Then
    ' Do nothing

  Else
    ' U.S.A.
    Text1.Text = Calendar1.Month & "/" & Calendar1.Day & "/" & Calendar1.Year
  End If

End Sub
```

This next bit of code goes under the option button labeled Europe:

```
Private Sub Option2_Click()

  ' Load the text box with the selected date
  If Text1.Text = "" Then
    ' Do nothing

  Else
    ' Europe
    Text1.Text = Calendar1.Day & "/" & Calendar1.Month & "/" & Calendar1.Year
  End If

End Sub
```

Press the *F5* key to run the program. As long as there are no compilation errors, the form will appear in the center of the screen. If you double-click on a date later than today's date, the date value will appear in the text box. However, if you double-click on a date earlier than today, a message will appear:

*** DateErr.bmp ***

By default, the date is placed into the text box in the format used in the U.S.A. Click on the option button labeled Europe and watch the contents of the text box change appropriately.

How It Works

First we declare the variables we need to accomplish this task. The first two need to be of the type Variant since they are used with the DateSerial statement:

```
' Declare the variables
  Dim SelDate As Variant, NowDate As Variant
  Dim SelYear As Integer, SelMonth As Integer, SelDay As Integer
```

Instead of simply retrieving the value of the selected date from the calendar, we'll retrieve it in pieces so we can correctly calculate the value for DateSerial:

```
' Retreive the date from the calendar control
With Calendar1
  SelYear = .Year
  SelMonth = .Month
  SelDay = .Day
End With
```

We then compute the serial value for the selected date.

```
` Compute the selected date
  SelDate = DateSerial(SelYear, SelMonth, SelDay)
```

Note that the DateSerial function will return an error if the value is -32,768 to 32,767 from the base date of January 1, 1904. This is important due to the range of years available from this control and the fact that they can be extended even further.

In a similar manner, we then compute today's date:

```
' Compute today's date
  NowDate = DateSerial((Format(Now, "yyyy")), (Format(Now, "m")), (Format(Now, "d")))
```

We need to check if the selected date is earlier than today. If it is, we'll generate an error to that effect. Once the message box is cleared, the calendar is refreshed with today's date so the user can have another go at it:

```
' Test to see if the selected date is earlier than today
  If SelDate < NowDate Then

    'If it is...
    Msg = "The  date you have selected is earlier than today." & Chr(10) & _
      "Please select today's date or later."
    Style = vbOKOnly + vbExclamation
    Title = "Date Selection Error"
    Response = MsgBox(Msg, Style, Title)

    ' Reset the calendar
    With Calendar1
      .Year = Format(Now, "yyyy")
      .Month = Format(Now, "m")
      .Day = Format(Now, "d")
      .Refresh
    End With
```

If the selected date is, in fact, today or later, we place the selected date into the text box in the appropriate format for the locale:

```
  Else
    ' Load the text box with the selected date
    If Option1.Value = True Then
      ' U.S.A.
      Text1.Text = SelMonth & "/" & SelDay & "/" & SelYear
    Else
      If Option2.Value = True Then
        ' Europe
        Text1.Text = SelDay & "/" & SelMonth & "/" & SelYear
        Else
          ' U.S.A.
          Option1.Value = True
          Text1.Text = SelMonth & "/" & SelDay & "/" & SelYear
      End If
    End If
  End If
```

The internationalization applied by the option buttons is rather simple, as it places the date fields into the text box in the correct order for the locale.

Notice that the controls on this form are located on a picture box, which acts as a container for them. We'll use this same sample to develop more concepts through the rest of this chapter. Later on, we'll be building a scheduling wizard from this form, so remember to keep track of where it is. However, this is where the other problem comes in …

Problem # 2 – The Picture Box

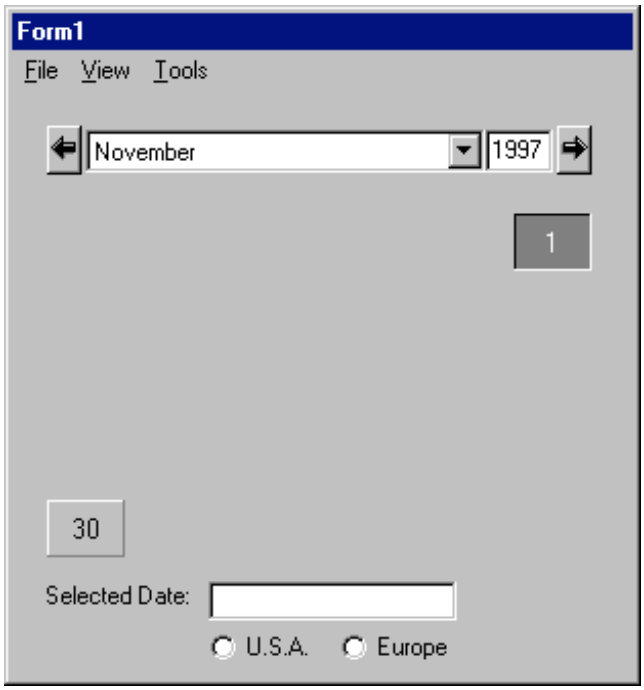
The Picture Box as a Container

You may be wondering why this form has a timer on the picture box. Then again, why would it need a picture box at all? The picture box has everything to do with wizards developed with Visual Basic's Wizard Manager. We need to see what the problem is and how to fix it before we get into the more complicated construction of a wizard.

To briefly explain, in a professional Windows application, a wizard is created using a single form, with the individual buttons (such as 'Back' and 'Next') having multiple uses. Off to the left of the screen is a stack of picture boxes, normally at `picBox.Left = -10000`. Each picture box contains a collection of controls for a single step of the wizard. When the wizard is started, only the picture box for the first step is on the form, while the rest are in the stack to the far left of the screen. When the 'Next' button is clicked, the picture box for the first step is swapped with the one for the second step. If the 'Back' button is pressed, the opposite occurs. The process with the 'Next' button continues until the 'Finish' button is clicked to close the wizard.

Unfortunately, the problem with the MSVBCalendar control only occurs when it has a picture box for a container. Remember, it worked just fine with its test program. Let's take a look at the problem in greater detail.

Go back into Chapter6.vbp and go to the properties for Timer1. You'll find that the Interval property is set to 1, which is in milliseconds. Change this value to 1000, or 1 second, and press F5. For 1 second, the form will look similar to this:



*** DateErr.bmp ***

Then it gets refreshed and looks as it should:

*** CalForm.bmp ***

Make sure you set the timer's interval back to 1 before continuing.

When I first started looking at this problem, I noticed something interesting. If I clicked within the current month when the calendar was in its incomplete state, all I got were more individual dates. But if I clicked on the previous or next months, the calendar recovered fully when it was repainted with the selected month.

I've added the timer to the form since simply refreshing the calendar upon Load or Activate didn't work. The timer is set for 1 millisecond, but its Enabled property is set to False at design-time. When the form is shown, the Activate event enables the timer:

```
Private Sub Form_Activate()

  ' Set the timer for its one-time firing
  Timer1.Enabled = True

End Sub
```

After the 1 millisecond time period has elapsed, the timer refreshes the calendar, then promptly disables itself:

```
Private Sub Timer1_Timer()

  ' Refresh the calendar so it appears correctly
  Calendar1.Refresh

  ' We're done, so disable the timer
  Timer1.Enabled = False
```

```
End Sub
```

This workaround becomes important for the development of wizards or when using a picture box as a container for other purposes. However, if you're using the MSVBCalendar on a form without using a picture box as a container, there's no need to implement this solution.

MSVBCalendar GUI Notes

Something to consider: We have the source code for this calendar control, so it should be quite a simple task to make changes as we see fit. We know how calendars work, as we've been using them most of our lives. Is there anything you would like to change about this calendar?

My first thought goes to the iteration buttons at the top of the calendar control itself. At first glance, it's a bit difficult to understand their exact purpose. As you may have noticed, the left one moves the calendar to the previous month, while the right one moves the calendar to the next month. However, the right iteration button is to the right of the Year text box. How many others think like I do and, at first, believed that this button on the right would affect the Year? A change to this aspect of the calendar might look something like this:

*** CldrNew.bmp ***

In designing user interfaces, it's important to place items correctly, in the locations the user will expect them to appear. Doing otherwise will cause unwarranted confusion. The user shouldn't require a Help file to use something as simple as a common calendar.

At this point, we're going to work on the menu structure of our form, looking at the resulting common dialogs differently then we ever have before.

Dialog Automation Objects

The Windows Common Dialogs, which are provided to us complements of commdlg32.dll, are quite important in the development of a GUI. All five of the common dialogs, Open (and Save), Color, Font, Print, and Help, are necessities which are well known to all computer users, regardless of their operating-system-of-choice. Each one of these dialogs fulfills a specific need for the user, and we, as developers, need to be aware of

exactly what the user is looking to do when he or she opens one of these dialogs in our applications.

One of the methods for implementing these dialogs in our applications is through the CommonDialog control. However, this control has always been a bit confusing to me. I've never quite found an easy way to keep track of all of the coding and flags necessary to implement a specific usage. Also, since we end up using the same control for various uses on different forms, the confusion can quickly snowball.

Microsoft has found a better way to implement the common dialogs from Visual Basic … without the CommonDialog control. All that's required is a single project reference to a rather small DLL, using syntax which is easy to understand. By taking the control off the form, and making the implementation of the common dialogs global to the project, a lot of the confusion subsequently disappears. The syntax has also been cleaned up a bit, without all of the cryptic flag settings.

The CommonDialog falls right out of the concept of code reuse with a loud thunk. Let's think about this for a moment. The CommonDialog control can be considered to be a template of sorts, and in using it we access the same template a number of times throughout a given application. But when else would we consciously place a windowless control on multiple forms and give it the same purpose in life each and every time, even duplicating the same code? A couple of aspects of the individual implementations may be different, but the redundancy can get old rather fast. Also, you're not always using all of the possible aspects of the control anywhere close to 100% of the time. So we end up with unused portions of the control, or "dead code", all through our applications.

In using a global definition for the common dialogs, it's kind of like placing the CommonDialog control on a module. We can get to it whenever we need it, only accessing the portion we need at the time. This is much more efficient.

In fact, for a couple of these dialogs we'll be placing the implementation in a module and accessing it whenever we need to, passing various items to new functions and retrieving the end result. We'll also discuss how to take care of the user's personal preferences, storing them in the registry and retrieving them as they are needed.

A dialog missing from the CommonDialog control is the Page Setup. This dialog has been included in the Dialog Automation Objects.

The only thing missing from the Dialog Automation Objects is the WinHelp dialog, and any functions related to it. You should use the WinHelp API for this anyway.

Setting Up The Dialog Automation Objects

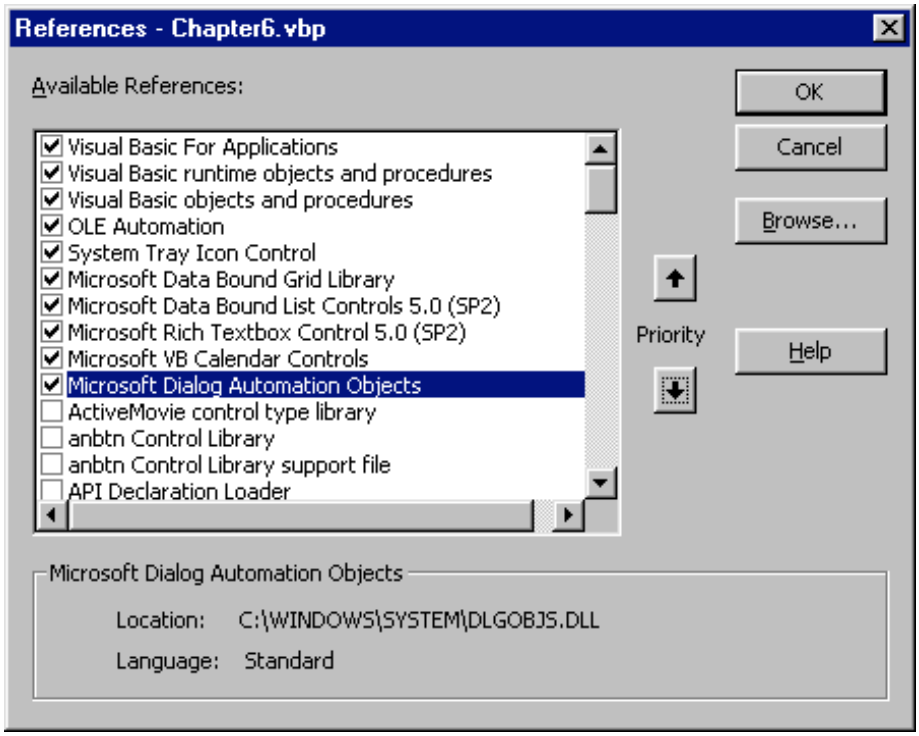
We need to install the Dialog Automation Objects before we can use them. Since there's no setup program for them, follow these steps:

1 Copy the file dlgobjs.dll from the **\Tools\Unsupprt\Dlgobj** directory on the Visual Basic CD to your **\Windows\System** directory. If you're installing this under Windows NT, copy dlgobjs.dll to the **\System32** directory.

2 While you're still looking at the contents of the CD, right click on the file **\Tools\Unsupprt\DlgObj\dlgobjs.reg** and left-click on 'Merge' from the popup menu. This registers the design-time license.

3 To register the DLL itself , select 'Run' from the Windows 'Start' menu. In the text box, type "**regsvr32.exe c:\windows\system\dlgobjs.dll**". Under Windows NT, this should be "**regsvr32.exe c:\windows\system32\dlgobjs.dll**". A message should appear telling you you've succeeded.

4 Back in the VB IDE, select "Microsoft Dialog Automation Objects" in the Project | References Dialog in Visual Basic. This makes the objects available to the current project.



*** DlgRef.bmp ***

Available Objects, Their Properties and Functions

Once we have the reference to Microsoft Dialog Automation Objects, we only need to open the Object Browser **DialogObjects** to view the various objects available to us. There are 5 dialog objects, each one giving us access to a specific dialog box. The available objects are:

ChooseColor
ChooseFile
ChooseFont
PageSetup
PrintDialog

There are also 5 helper objects which are used by the dialog objects:

Colors
FileNames
Filters
PrinterDevice
Rectangle

We'll be looking at the dialog objects individually, and within those discussions, we'll also look at how the helper objects are used.

The Dialog Objects

In order to use any of the Dialog objects, we need to create an instance of the object we want to use. This can be accomplished quite simply using the Set statement:

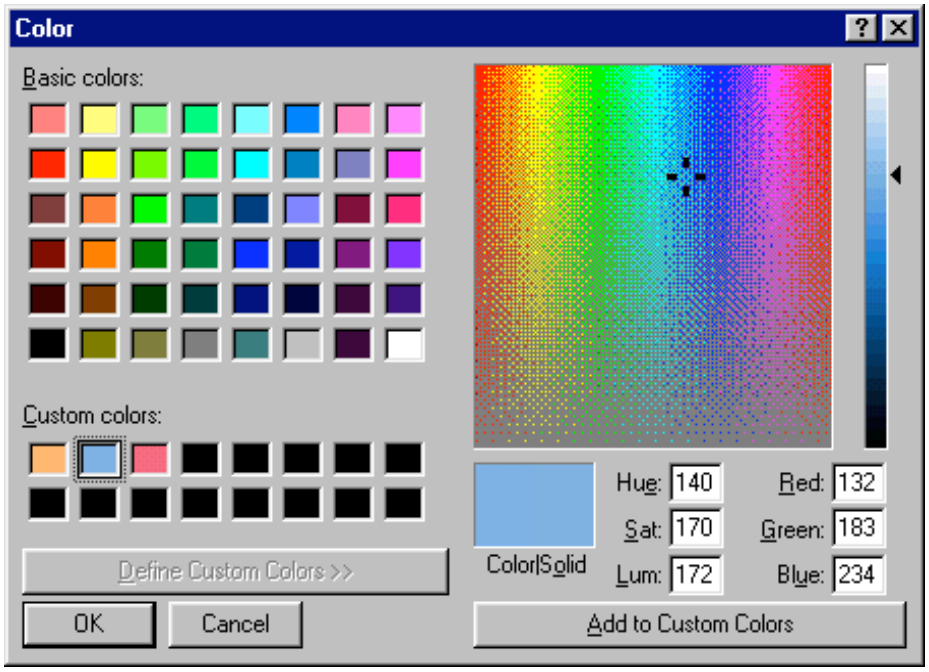
```
Set ChooseFile1 = New ChooseFile
```

Now we can use any of the properties and functions of the new object, in this case, `ChooseFile1`.

We're going to be looking at these dialogs in their order of difficulty, starting with the Color Palette, and continuing through to the Font dialog.

ChooseColor

The ChooseColor object implements the Color Palette dialog, allowing the user to change the color of something in an application. This dialog is implemented with all of its custom color functions intact, and can even be opened with custom colors being loaded from somewhere else, either a set list or from the registry:



*** ClrPalet.bmp ***

The object's properties and single function are rather easy to understand, due to some correctly-named properties:

Property/Function	Description
Property **Center** As Boolean	Centers the dialog on the screen
Property **Color** As Long	The currently selected color, or the one to start out with
Property **CustomColors** As Colors	A collection of up to 16 custom colors
Property **hWnd** As Long	The window this dialog will act modally against, with 0 being the desktop
Property **PreventCustomColors** As Boolean	Disables the "Define Custom Colors>>" button
Property **ShowCustomColors** As Boolean	Opens the dialog with the custom colors extended
Function **Show**() As Boolean	Shows the dialog

The helper object named Colors is a collection of custom colors used by the CustomColor property of the ChooseColor object. This helper object has two properties:

Property	Description
Property **Count** As Long	Provides a count of the custom colors
Property **Item***(Index As Long)* As Long	Provides an index of the individual custom colors

At this point in each of the following descriptions, we're going to be looking at examples of the dialogs. Each of the examples will listing each property or function at least once. This way, you can see a little clearer how to use each one.

Try It Out – Calling The Color Palette

This is the first dialog we'll be calling from a function in a module. In doing so, we can call it more than once without having to rewrite it each time. This is particularly important when it comes to custom colors. Otherwise, we end up with a number of custom color arrays in the registry, one for each form we call the Color dialog from.

Add a module to the project Chapter6.vbp and add the following function to it:

```
Public Function SelectColor(ColorNumber As Long) As Long

    Dim n As Integer, i As Integer

    ' Create an instance of the ChooseColor object
    Set ChooseColor1 = New ChooseColor

    With ChooseColor1
      ' Get the custom colors from the registry
      For n = 0 To 15
        .CustomColors(n) = GetSetting("Chapter6", _
          "Startup", "CustomColors(" & n & ")", "0")
      Next n

      ' Center the dialog on the screen
      .Center = True
```

```
    ' Set the initial color
    .Color = ColorNumber

    ' The window this dialog will operate
    ' modally against (0 = desktop)
    .hWnd = 0

    ' Indicate whether or not the user can
    ' expand the custom colors
    .PreventCustomColors = False

    ' Whether or not the custom colors are
    ' expanded by default
    .ShowCustomColors = False

    ' Show the dialog
    .Show

    ' Write the custom colors to the registry
    For i = 0 To 15
      SaveSetting "Chapter6", "Startup", _
        "CustomColors(" & i & ")", .CustomColors(i)
    Next i

  End With

  ' Return the selection
  SelectColor = ChooseColor1.Color
End Function
```

This function creates a new ChooseColor object, then loads the 16 custom colors from the registry. If the registry settings don't exist yet, all of the individual colors are set to "0", or black. The dialog is centered on the screen, and the initial color is set to the variable "ColorNumber". This is the variable that is passed to this function from the procedure calling it, and is the current color of the item to have its color changed. Three other properties are set before showing the form. The hWnd property is set to 0 so the Color dialog acts modally against the desktop. The PreventCustomColors is set to False. Doing so **allows** the custom colors to be shown, enabling the "Define Custom Colors>>" button. ShowCustomColors is also set to False, meaning the custom colors are not extended when the dialog is first displayed. Then, the Color dialog is finally shown.

The function continues once the Color dialog is closed. Immediately, the custom colors are written to the registry. Then, the newly selected color is returned to the calling procedure and the function ends.

Try It Out - Implementing The SelectColor Function

Add the following code to the Load event of Form1 in Calendar.vbp:

```
Private Sub Form_Load()

  SaveSetting "Chapter6", "Startup", "TestSetting", "Test"

  With Calendar1
    ' Set the calendar to today's date
    .Year = Format(Now, "yyyy")
    .Month = Format(Now, "m")
    .Day = Format(Now, "d")

    ' Set the colors
    .DayColor = GetSetting("Chapter6", "Startup", "DayColor", "0")
    .DayNameColor = GetSetting("Chapter6", "Startup", "DayNameColor", "0")

    ' Update the settings
```

```
    .Refresh
  End With

End Sub
```

This will retrieve the colors we save to the registry for the calendar's DayColor and DayNameColor properties. Using the Refresh method ensures the properties are updated every time the calendar is opened. This routine also writes a test setting to the registry, which we'll be discussing shortly.

Add this code to the mnuViewColorDayColor_Click event:

```
Private Sub mnuViewColorDayColor_Click()

  ' Set the calendar's DayColor property via the Color dialog
  Calendar1.DayColor = SelectColor(Calendar1.DayColor)

  ' Save the DayColor property to the registry
  SaveSetting "Chapter6", "Startup", "DayColor", Calendar1.DayColor

End Sub
```

To ensure the same functionality is available for the calendar's day names, add the following code to the mnuViewColorDayNameColor_Click event:

```
Private Sub mnuViewColorDayNameColor_Click()

  ' Set the calendar's DayNameColor property via the Color dialog
  Calendar1.DayNameColor = SelectColor(Calendar1.DayNameColor)

  ' Save the DayNameColor property to the registry
  SaveSetting "Chapter6", "Startup", "DayNameColor", Calendar1.DayNameColor

End Sub
```

Now, add this code to the click event of the mnuToolsRestoreDefaults menu item:

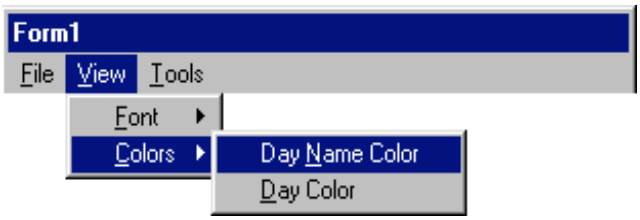
```
Private Sub mnuToolsRestoreDefaults_Click()

  Dim TestItem As String

  ' Check the test setting
  TestItem = GetSetting("Chapter6", "Startup", "TestSetting", "")

  ' If the test setting is clear, the settings don't exist
  If TestItem = "" Then
    ' Do Nothing

  Else
    ' Clear the settings if there are any
    DeleteSetting "Chapter6", "Startup"
  End If

  With Calendar1

    ' Reset the colors to black
    .DayColor = 0
    .DayNameColor = 0

  End With

End Sub
```

This last routine erases any of the registry settings we've previously written in this section. This way, remnants from this chapter aren't taking up space in your registry forever! It also restores the color properties to their default setting of "black" (0).

This is also where the TestSetting in the registry is used, which was mentioned a short while ago. If the test setting is returned as an empty string, this indicates this area of the registry is empty and the DeleteSettings procedure is not called (if it were called, an error would result). Otherwise, the DeleteSettings procedure deletes the entire section.

The individual procedures mnuViewColorDayColor_Click() and mnuViewColorDayNameColor_Click() save their particular setting to the registry. This would be difficult to do in the SelectColor function without passing a number of parameters to it.

Press F5, or select Start from Visual Basic's Run menu. Click on the Day Name Color menu item:



*** DayName.bmp ***

The Color dialog will appear, with the black color selected, the custom colors all set to black, and the custom color selection area not extended. Change the current selection, making sure to load up some of the 16 custom colors boxes with new colors. Once you close the dialog, the calendar's day names will be changed to the selected color.

Now select Day Color from the form's menu. When the Color dialog appears, the custom colors will be loaded with the ones you'd previously selected. Select a new color and close the Color dialog. The colors of the days in the calendar grid will be changed to the new selection.

Finally, click Restore Default Settings from the Tools menu. The DayColor and DayName color properties will return to black.

Color Palette GUI Notes

The selected color is normally assigned to a particular item in our application. Keeping in mind that the user may have selected some overall Windows color scheme having some semblance of weirdness, we really don't want to assign custom colors to controls. In this case, we need to open the Color Palette dialog without the custom colors being extended, and having the "Define Custom Colors>>" button being disabled.

Note that the menu items are designed with Day Name Color being above Day Color. This follows the layout of the calendar itself, where the day names are at the top of the grid. In the code for the calendar, these items were seen in reverse order from this due to alphabetization by Visual Basic and whoever originally wrote the code. It would be easy to accidentally design the menu items to follow the structure of the code, which would be confusing to the user. Be sure to place first things first and last things last as the user will see them to prevent this kind of confusion.

How It All Works

Since the Dialog Automation Objects are not supplied as source code, we need to determine how they are were developed, in case they're not supplied to us in the future. The API Viewer on the Add-In's menu in the IDE provides this information.

The Dialog Automation Objects are implemented as an ActiveX DLL. The individual dialog objects are declared via the Windows API, such as the following declaration for ChooseColor:

```
Declare Function ChooseColor Lib "comdlg32.dll" Alias "ChooseColorA" _
 (pChoosecolor As CHOOSECOLOR) As Long
```

The type CHOOSECOLOR is declared in the same area of the DLL's source code as the above API declaration:

```
Type CHOOSECOLOR
        lStructSize As Long
        hwndOwner As Long
        hInstance As Long
        rgbResult As Long
        lpCustColors As Long
        flags As Long
        lCustData As Long
        lpfnHook As Long
        lpTemplateName As String
End Type
```

Using this information, the entire ChooseColor class, including the individual properties, is developed. The properties we're using fill in the blanks in the CHOOSECOLOR type. The **ChooseColor.Show** property we've used in our own code calls the API through the ChooseColorA function, which then uses the CHOOSECOLOR type, and everything is set up correctly to display the Color dialog.

ChooseFile

The ChooseFile object gives us access to both the Open and Save As dialog boxes. We'll go ahead and take a look at both of these dialogs so we can be clear on the differences between the two implementations.

There are a few more properties for this object than for ChooseColor, but still only the one **.Show** function:

Property/Function	**Description**
Property **Center** As Boolean	Centers the dialog on the screen
Property **Directory** As String	The directory to open, or the chosen directory
Property **FileMustExist** As Boolean	Whether or not the file must exist to be chosen (Open dialog only)
Property **FileName** As String	The name of the file to look for, or the file selected
Property **FileNames** As FileNames	A collection of file names in a multiselect dialog
Property **Filters** As Filters	A collection of file extension filters
Property **HideReadOnly** As Boolean	Whether or not to show the Read-only checkbox (Open dialog only)
Property **hWnd** As Long	The window the dialog will act modally against, with 0 being the desktop

Property **MultiSelect** As Boolean	Whether or not the dialog permit the selection of multiple files
Property **OverwritePrompt** As Boolean	Whether or not the system will prompt the user prior to overwriting a file
Property **ReadOnly** As Boolean	True if the file is Read-only
Property **Save** As Boolean	True to show the Save dialog, False for Open
Property **Title** As String	The new title of the dialog
Function **Show**() As Boolean	Shows the dialog

Two more of the helper objects are used by ChooseFile. The first one is FileNames, and is used for multiple file selections:

FileNames

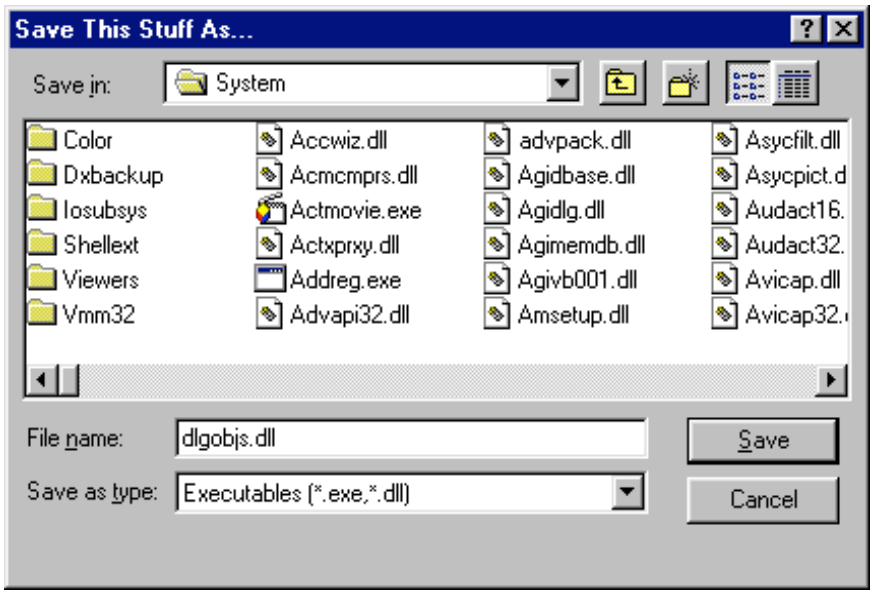
Property	**Description**
Property **Count** As Long	A count of the multiple file names
Property **Item***(Index As Long)* As String	The index of each individual file name

The second helper object is Filters, which we'll use to load the Type drop-down box in the dialog via a collection of Filter items:

Filters

Property	**Description**
Property **Count** As Long	A count of the filters in the collection
Property **Item***(Index As Long)* As String	The index of each individual filter
Sub **Add***(bstrNew As String)*	Adds a filter to the collection
Sub **Remove***(Index As Long)*	Remove a filter from the collection

Try It Out – The "Save As" Dialog



*** SaveAs.bmp ***

There are quite a few items in this dialog which will change depending on the specific usage. The directory is one item, as are the file name, filters, title, and multiple file selections. It makes more sense to place individualized code under each Save As menu item than to have a common function and pass all of these variables to and from the function.

Place the following code under the mnuFileSaveAs_Click event in our Chapter6 sample:

```
Private Sub mnuFileSaveAs_Click()
```

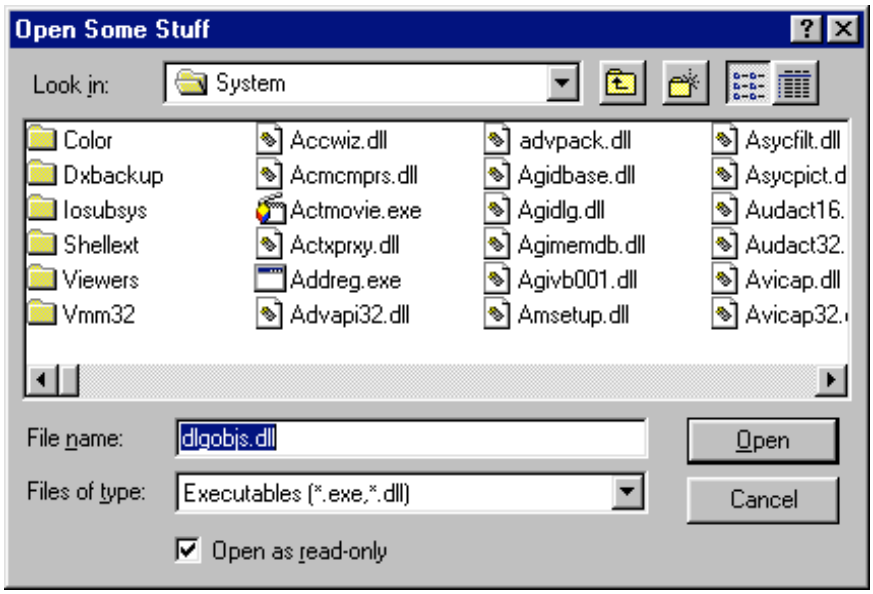
```
  ' Create the new object
  Set ChooseFile1 = New ChooseFile

  With ChooseFile1

    ' Center the dialog
    .Center = True

    ' Specify the default directory
    .Directory = GetSetting("Chapter6", "Startup", _
      "SaveAsDirectory", "c:\windows\system\")

    ' Does not apply to the Save dialog
    '.FileMustExist = True

    ' File to be initially selected
    .filename = "dlgobjs.dll"

    ' Load the Filter drop-down box
    With .Filters
      .Add "Executables (*.exe,*.dll):*.exe;*.dll"
      .Add "Documents (*.doc,*.txt):*.doc;*.txt"
      .Add "Stuff (*.stf):*.stf"
      .Add "All Files (*.*):*.*"
    End With

    ' Does not apply to the Save dialog
    '.HideReadOnly = True

    ' The window this dialog will operate
    ' modally against (0 = desktop)
```

```
    .hWnd = 0

    ' Allow the user to select more than one file
    .MultiSelect = True

    ' Prompt before overwriting a file
    .OverwritePrompt = True

    ' Does not apply to Save As
    '.ReadOnly = False

    ' Save?  Or Open?
    .Save = True

    ' Retitle the dialog
    .Title = "Save This Stuff As..."

    ' Show the dialog
    .Show

    ‘ Save the new default directory
    SaveSetting "Chapter6", "Startup", "SaveAsDirectory", .Directory

    If (.filename = "") Then
      ' Handler in case 'Cancel' is selected
      Exit Sub
    Else
      ' Place save code here
    End If

  End With
End Sub
```

Since we’re not actually saving anything at this point, toward the bottom you’ll see “Place save code here”. We’ll replace this comment with code to save something in a later chapter.

Start the Chapter6 project and select Save As from the File menu. The Save As dialog will appear as it was shown at the beginning of this section. Change the directory to something different than the default of c:\windows\system\, which is given in the GetSettings statement. Close the dialog, then open it again. Since the SaveSettings statement wrote the directory you chose to the registry, this is the directory which is selected when you open the dialog again.

Try It Out – The “Open” Dialog



*** Open.bmp ***

The Open dialog is almost identical to the Save As dialog in its implementation. Place the following code in the mnuFileOpen_Click event:

```
Private Sub mnuFileOpen_Click()

  ' Create the new object
  Set ChooseFile1 = New ChooseFile

  With ChooseFile1

    ' Center the dialog
    .Center = True

    ' Specify the default directory
    .Directory = GetSetting("Chapter6", "Startup", _
      "OpenDirectory", "c:\windows\system\")

    ' Does not apply to the Save dialog
    .FileMustExist = True

    ' File to be initially selected
    .filename = "dlgobjs.dll"

    ' Load the Filter drop-down box
    With .Filters
      .Add "Executables (*.exe,*.dll):*.exe;*.dll"
      .Add "Documents (*.doc,*.txt):*.doc;*.txt"
      .Add "Stuff (*.stf):*.stf"
      .Add "All Files (*.*):*.*"
    End With

    ' Hide the Read-only checkbox?
    .HideReadOnly = False

    ' The window this dialog will operate
    ' modally against (0 = desktop)
    .hWnd = 0

    ' Allow the user to select more than one file
    .MultiSelect = True

    ' Prompt before overwriting a file
```

```
    .OverwritePrompt = True

    ' Read-only checkbox checked?
    .ReadOnly = True

    ' Save?  Or Open?
    .Save = False

    ' Retitle the dialog
    .Title = "Open Some Stuff"

    ' Show the dialog
    .Show

    ' Save the new default directory
    SaveSetting "Chapter6", "Startup", "OpenDirectory", .Directory

    If (.filename = "") Then
      ' Handler in case 'Cancel' is selected
      Exit Sub
    Else
      ' Place open code here
    End If
  End With
End Sub
```

Notice the same point about the directory as in the Save As dialog. The GetSetting and SaveSetting statements are, in fact, identical in both cases, except for the registry key being "SaveAsDialog" in one instance, and "OpenDialog" in the other.

How It Works

After creating the new dialog object ChooseFile1, we make sure it will be centered on the screen. The GetSetting statement does its job to get the default directory, and we specify that the file must exist (which seems strange since we're looking at a current list of the directory!) Following the naming of the file we're looking for, we lay out the collection of filters, making sure to include the filter for "All Files *.*". The HideReadOnly property is set to False so the Read-only check box will be shown. The hWnd property is set as in the Color dialog, and we also set MultiSelect to True so the user can select more than one file. Setting OverwritePrompt to True provides a message box to appear before the user overwrites a file that already exists. The Save property sets the dialog for Save or Open functionality, and we also retitle the dialog to something appropriate to the application.

After the dialog is closed, SaveSetting is used to write the directory to the registry, and the procedure finishes with the open statement we'll fill in later.

PageSetup

The Page Setup dialog is an extra dialog available from the Dialog Automation Objects. Since this dialog is not available from the CommonDialog control, we'll do a little extra with it.

The properties are, once again, appropriately named for their functions. Notice there are fewer properties for this dialog than even for the Color dialog:

Property/Function	**Description**
Property **Center** As Boolean	Centers the dialog on the screen

Property **DisableMargins** As Boolean	Whether or not the Margins section of the dialog is disabled
Property **DisableOrientation** As Boolean	Whether or not the Orientation section of the dialog is disabled
Property **DisablePaper** As Boolean	Whether or not the Paper section of the dialog is disabled
Property **DisablePrinter** As Boolean	Whether or not the Printer section of the dialog is disabled
Property **hWnd** As Long	The handle of the window this dialog will act modally against, with 0 being the desktop
Property **Margins** As Rectangle	The margins set by the user, relative to the top and left edges of the paper
Property **MinimumMargins** As Rectangle	The minimum margins allowed, relative to the top and left edges of the paper
Property **PreventWarning** As Boolean	Whether or not a message will appear if no default printer is selected
Property **PrinterDevice** As PrinterDevice	The actual printer device (see below)
Function **Show**() As Boolean	Shows the dialog

Printer Device

The PrinterDevice object is used for returning the specifics for the printer to be used. At the end of our sample procedure, we'll enumerate this to see exactly what the values become for our selection:

Property	**Description**
Property **Copies** As Integer	Number of copies to be printed
Property **Default** As Boolean	Whether or not the selected printer is the default
Property **Driver** As String	The OEM name of the printer driver
Property **DriverVersion** As Integer	The driver's version number
Property **Name** As String	Common name for the selected printer
Property **Orientation** As Integer	1 = portrait, 2 = landscape
Property **Output** As String	Port to be printed to (i.e., LPT1) or "FILE"
Property **PaperSize** As Integer	Size of the paper, as designated by the OEM (i.e., Hewlett-Packard 8 _ x 11 inch = 1)

Rectangle

The Rectangle object is used to set the MinimumMargin and Margin properties, relative to the edge of the selected paper size:

Property **Bottom** As Long

Property **Left** As Long

Property **Right** As Long

Property **Top** As Long

Try It Out – The Page Setup Dialog

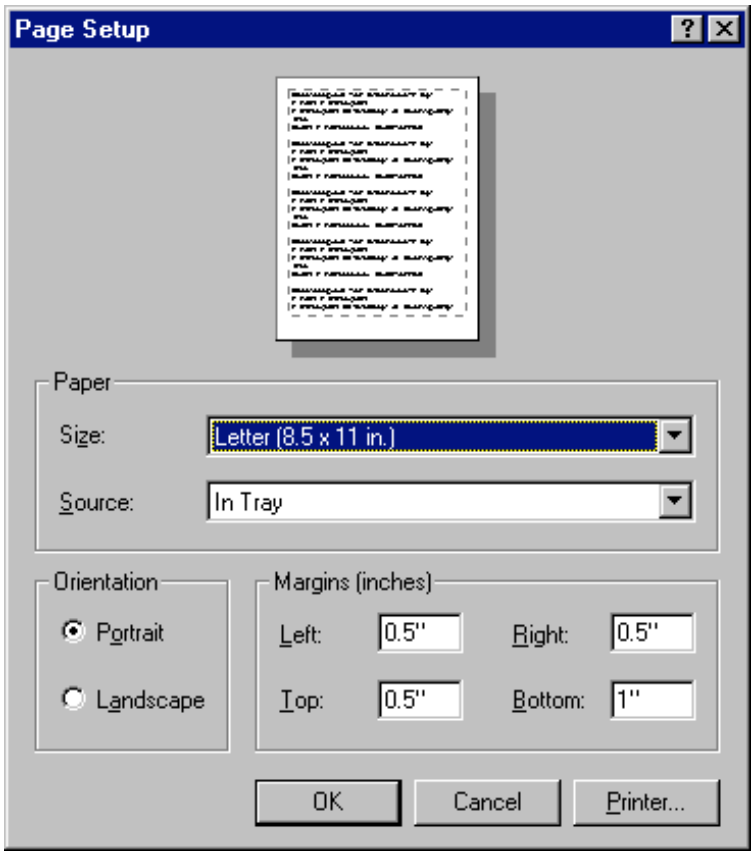
Add the following code to the mnuFilePageSetup_Click event of the form in Chapter6.vbp:

```
Private Sub mnuFilePageSetup_Click()
  ' Create the new object
  Set PageSetup1 = New PageSetup

  With PageSetup1

    ' Center the dialog
    .Center = True

    ' Set the Disable properties
    .DisableMargins = False
    .DisableOrientation = False
    .DisablePaper = False
    .DisablePrinter = False

    ' Window the dialog will act
    ' modally against (0 = desktop)
    .hWnd = 0

    'Set the margins (in 1/1000ths of an inch)
    With .Margins
      .Top = 500
      .Left = 500
      .Right = 500
      .Bottom = 1000
    End With

    'Set the minimum margins
    ' (in 1/1000ths of an inch)
    With .MinimumMargins
      .Top = 250
      .Left = 250
      .Right = 250
      .Bottom = 750
    End With

    ' Suppress "No default printer" warning
    .PreventWarning = False

    ' Show the dialog
    .Show

    If (.PrinterDevice.Name = "") Then

      ' Handler if Cancel is selected
      Exit Sub

    Else
      With .PrinterDevice
      ' Return the selected settings
        MsgBox "You selected:" & Chr(10) & _
          Chr(9) & "Copies: " & .Copies & Chr(10) & _
          Chr(9) & "Default: " & .Default & Chr(10) & _
          Chr(9) & "Driver: " & .Driver & Chr(10) & _
          Chr(9) & "Driver Version: " & .DriverVersion & Chr(10) & _
          Chr(9) & "Printer Name: " & .Name & Chr(10) & _
          Chr(9) & "Orientation: " & .Orientation & Chr(10) & _
          Chr(9) & "Output: " & .Output & Chr(10) & _
          Chr(9) & "Paper Size: " & .PaperSize & Chr(10) & _
          Chr(9) & "Top Margin: " & PageSetup1.Margins.Top & Chr(10) & _
          Chr(9) & "Left Margin: " & PageSetup1.Margins.Left & Chr(10) & _
          Chr(9) & "Right Margin: " & PageSetup1.Margins.Right & Chr(10) & _
          Chr(9) & "Bottom Margin: " & PageSetup1.Margins.Bottom
      End With
    End If
```

```
  End With
End Sub
```

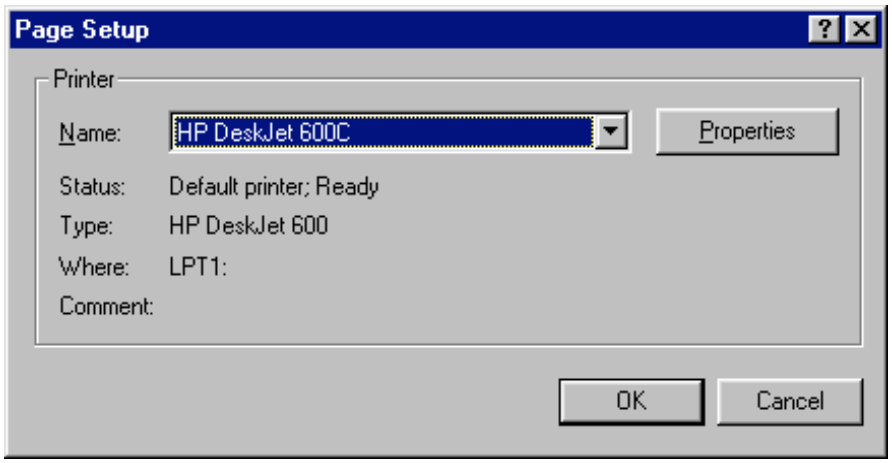
This isn't a very complicated procedure, but the list of information it returns is quite large. Because of this, we can't very well create a function out of it in any practical manner.

When you start the project from the Run menu and select File | Page Setup, the dialog box will appear as specified:



*** PgSetup.bmp ***

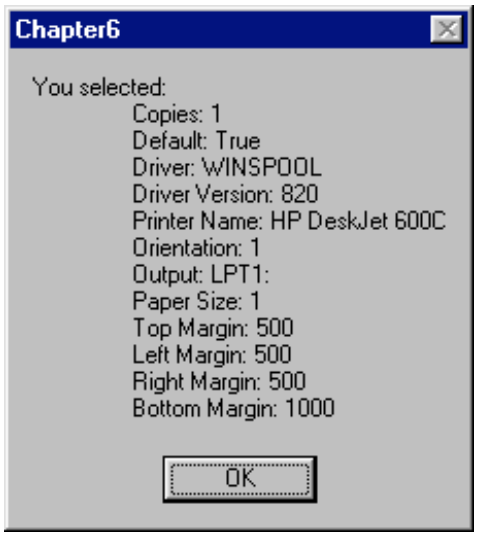
Go through this dialog and make changes to its settings. This dialog acts in the same way as it does in Microsoft Word and other Windows 95 programs. If you change the Margin settings, you'll notice that it's not possible to set the margins smaller than the MinimumMargin property settings. Also, the Printer button shows the PrinterSelection dialog:



*** PntSetup.bmp ***

Selecting a printer is as simple as making a new selection from the Name dropdown box. Pressing the Properties button calls the OEM property page for the selected printer.

When you close all the other dialogs and select the OK button on the Page Setup dialog, a message box will appear, showing all of the selections we made, which are necessary for the printing of a document:



*** SetupMsg.bmp ***

This information is then used to setup whatever document we're going to print.

A few of these items seem rather cryptic and need a bit of explanation. For instance, the WINSPOOL driver indicates we're spooling the printing through the Windows Print Manager. The orientation is 1 for portrait, whereas 2 would be for landscape. And Paper Size? For this printer, there are a large number of sizes. 1 is for letter size, or 8 _ x 11 inches, 5 is for legal size, which is 8 _ x 14 inches, 9 gives us A4 size paper, being 210 x 297 mm, and so on.

PrintDialog

The Print dialog is quite similar to the Page Setup dialog in many respects, to the point where it uses the same type of PrinterDevice object to relay information about the printer. It does have a few more properties to get the job done:

Property/Function	Description
Property **Center** As Boolean	Centers the dialog on the screen
Property **DisablePageNumbers** As Boolean	Disables the Page Numbers section of the dialog
Property **DisablePrintToFile** As Boolean	Disables the Print To File section of the dialog
Property **DisableSelection** As Boolean	Disables the Page Selection section of the dialog
Property **hWnd** As Long	Handle of the window this dialog will act modally against, with 0 being the desktop
Property **MaxPage** As Long	Maximum page the user can select for printing
Property **MinPage** As Long	Minimum page the user can select for printing
Property **PreventWarning** As Boolean	Whether or not a message will appear if no default printer is selected
Property **PrinterDevice** As PrinterDevice	The actual printer device (see description in the Page Setup section)
Property **PrintRange** As Long	Declaration of which pages to print
Property **PrintToFile** As Boolean	Whether or not to print to a file
Property **ShowPrintToFile** As Boolean	Whether or not to show the Print To File check box
Function **Show**() As Boolean	Shows the dialog

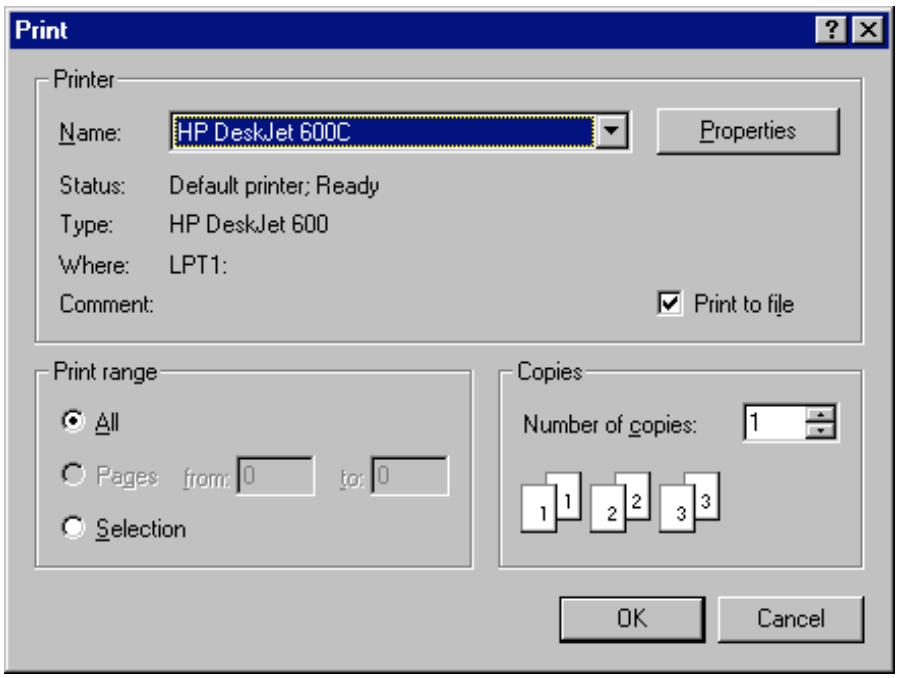
Try It Out – The Print Dialog

Place the following code under the mnuFilePrint_Click event on the form in our Chapter6 sample:

```
Private Sub mnuFilePrint_Click()

' Create the new object
 Set PrintDialog1 = New PrintDialog

 With PrintDialog1

   ' Center the dialog
   .Center = True

   ' Set the Disable properties
   .DisablePageNumbers = False
   .DisablePrintToFile = False
   .DisableSelection = False

   ' Window the dialog will act
   ' modally against (0 = desktop)
   .hWnd = 0

   ' Suppress "No default printer" warning
```

```
    .PreventWarning = False

    ' Set the Print To File characteristics
    .PrintToFile = True
    .ShowPrintToFile = True

    .PrintRange = 2

    ' Show the dialog
    .Show

    If (.PrinterDevice.Name = "") Then

      ' Handler if Cancel is selected
      Exit Sub

    Else

      ' Place print code here

    End If
  End With
End Sub
```

When you start the Chapter6 sample from the Visual Basic IDE and click the File | Print command, you'll see this dialog box:



*** PrintDlg.bmp ***

Compared with some of the other dialogs in this group, this one was rather simple.

Following is the discussion for the Font dialog, likely the most complicated one of them all.

ChooseFont

As complex as the Font dialog can be, we can still implement it as a function and call the single function from anywhere in our program. This is accomplished through the use of the StdFont object, which encapsulates all of the properties of a font into a single object. All we need to do is call the function that calls the Font dialog, and the font is returned as a StdFont object.

The Font dialog is implemented without any helper objects … except the IFontDisp type for the GetFont() function. We don't need to worry about this one since Visual Basic takes care of it through OLE:

Property/Function	**Description**
Property **Center** As Boolean	Centers the dialog on the screen
Property **Color** As Long	The color to be applied to the font
Property **FaceName** As String	Name of the font (i.e., "MS Sans Serif")
Property **FixedPitchOnly** As Boolean	If True, only shows fixed-pitch fonts
Property **ForceFontExist** As Boolean	Whether or not the font must exist
Property **Height** As Long	Height of the font (i.e., 10)
Property **hWnd** As Long	Handle of the window this dialog will act modally against, with 0 being the desktop
Property **Italic** As Boolean	Whether or not the font is *italicized*
Property **ShowEffects** As Boolean	If True, show the Effects section of the dialog
Property **ShowPrinterFonts** As Boolean	Whether or not to show printer fonts in the listing
Property **ShowScreenFonts** As Boolean	Whether or not to show screen fonts in the listing
Property **ShowVerticalFonts** As Boolean	Whether or not to show vertical fonts in the listing
Property **SizeMax** As Long	Maximum size allowed (also refered to as Height)
Property **SizeMin** As Long	Minimum size allowed (also refered to as Height)
Property **StrikeOut** As Boolean	Whether or not the selected font has the ~~Strikeout~~ effect applied
Property **TrueTypeOnly** As Boolean	If True, only shows True Type fonts
Property **Underline** As Boolean	Whether or not the selected font has the <u>Underline</u> effect applied
Property **Width** As Long	Width of the font, for style (i.e., 400 = Regular, 700 = **Bold**, etc.)
Function **GetFont()** As IFontDisp	Gets the selected font from the dialog as a Font object
Function **Show()** As Boolean	Shows the dialog

The ShowEffects setting must be set to True prior to setting the StrikeOut, Underline, or Color effects. Otherwise, an error will occur.

Try It Out – Using The Font Dialog

Add the following function to the module we created earlier in the Chapter6 project:

```
Public Function SelectFont() As StdFont

  Dim NewFont As New StdFont

  'Create the new object
  Set ChooseFont1 = New ChooseFont

  With ChooseFont1

    ' Center the dialog on the screen
    .Center = True

    ' Window the dialog will act
    ' modally against (desktop = 0)
    .hWnd = 0

    ' Set the dialog properties
    .FixedPitchOnly = False
    .ShowPrinterFonts = True
    .ShowScreenFonts = True
    .ShowVerticalFonts = True

    ' Show Effects selections
    .ShowEffects = True

    ' Set whether or not to display an error
    ' message if the selected font does not
    ' exist (i.e., no selection made = Error)
    .ForceFontExist = True

    ' Set the Size properties
    .SizeMin = 10
    .SizeMax = 36

    .StrikeOut = False
    .Underline = False

    ' Set the Color property
    .Color = 0

    ' Show the dialog
    .Show

    ' After the dialog is closed,
    ' change the label to the selected font
    With NewFont
      .Bold = ChooseFont1.GetFont.Bold
      .Charset = ChooseFont1.GetFont.Charset
      .Italic = ChooseFont1.GetFont.Italic
      .Name = ChooseFont1.GetFont.Name
      .Size = ChooseFont1.GetFont.Size
      .Strikethrough = ChooseFont1.GetFont.Strikethrough
      .Weight = ChooseFont1.GetFont.Weight
    End With
  End With

  Set SelectFont = NewFont

End Function
```

Place the following code under the mnuViewFontDayNameFont_Click event on the form:

```
Private Sub mnuViewFontDayNameFont_Click()

  ' Creat a new font object
```

```
  Dim NewDayNameFont As New StdFont

  ' Set the new font object via the Font dialog
  Set NewDayNameFont = SelectFont

  ' Update the calendar's DayNameFont property
  Set Calendar1.DayNameFont = NewDayNameFont

  ' Write the new DayNameFont property to the registry
  SaveSetting "Chapter6", "Startup", "DayNameFontBold", Calendar1.DayNameFont.Bold
  SaveSetting "Chapter6", "Startup", _
    "DayNameFontCharset", Calendar1.DayNameFont.Charset
  SaveSetting "Chapter6", "Startup", _
    "DayNameFontItalic", Calendar1.DayNameFont.Italic
  SaveSetting "Chapter6", "Startup", _
    "DayNameFontName", Calendar1.DayNameFont.Name
  SaveSetting "Chapter6", "Startup", "DayNameFontSize", Calendar1.DayNameFont.Size
  SaveSetting "Chapter6", "Startup", _
    "DayNameFontStrikethrough", Calendar1.DayNameFont.Strikethrough
  SaveSetting "Chapter6", "Startup", _
    "DayNameFontUnderline", Calendar1.DayNameFont.Underline

  ' Update the calendar
  Calendar1.Refresh
End Sub
```

Place the following code under the mnuViewFontDayFont_Click event on the form:

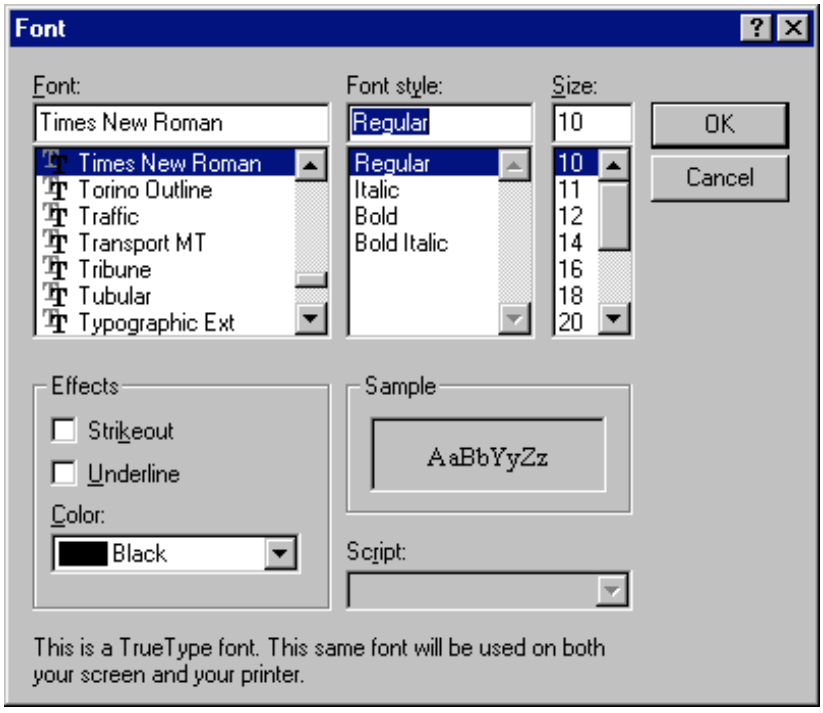
```
Private Sub mnuViewFontDayFont_Click()

  ' Create a new font object
  Dim NewDayFont As New StdFont

  ' Set the new font object via the Font dialog
  Set NewDayFont = SelectFont

  ' Update the calendar's DayFont property
  Set Calendar1.DayFont = NewDayFont

  ' Write the new DayFont property to the registry
  SaveSetting "Chapter6", "Startup", "DayFontBold", Calendar1.DayFont.Bold
  SaveSetting "Chapter6", "Startup", "DayFontCharset", Calendar1.DayFont.Charset
  SaveSetting "Chapter6", "Startup", "DayFontItalic", Calendar1.DayFont.Italic
  SaveSetting "Chapter6", "Startup", "DayFontName", Calendar1.DayFont.Name
  SaveSetting "Chapter6", "Startup", "DayFontSize", Calendar1.DayFont.Size
  SaveSetting "Chapter6", "Startup", _
    "DayFontStrikethrough", Calendar1.DayFont.Strikethrough
  SaveSetting "Chapter6", "Startup", "DayFontUnderline", Calendar1.DayFont.Underline

  ' Update the calendar
  Calendar1.Refresh

End Sub
```

Finally, change the forms Load event by adding the following code:

```
Private Sub Form_Load()

  SaveSetting "Chapter6", "Startup", "TestSetting", "Test"

  With Calendar1
    ' Set the calendar to today's date
    .Year = Format(Now, "yyyy")
    .Month = Format(Now, "m")
    .Day = Format(Now, "d")

    ' Set the colors
    .DayColor = GetSetting("Chapter6", "Startup", "DayColor", "0")
    .DayNameColor = GetSetting("Chapter6", "Startup", "DayNameColor", "0")
```

```
    ' Retrieve the calendar's DayFont settings from the registry
    With .DayFont
      .Bold = GetSetting("Chapter6", "Startup", "DayFontBold", "False")
      .Charset = GetSetting("Chapter6", "Startup", "DayFontCharset", "77")
      .Italic = GetSetting("Chapter6", "Startup", "DayFontItalic", "False")
      .Name = GetSetting("Chapter6", "Startup", "DayFontName", "MS Sans Serif")
      .Size = GetSetting("Chapter6", "Startup", "DayFontSize", "10")
      .Strikethrough = GetSetting("Chapter6", "Startup", _
        "DayFontStrikethrough", "False")
      .Underline = GetSetting("Chapter6", "Startup", "DayFontUnderline", "False")
    End With

    ' Retrieve the calendar's DayNameFont settings from the registry
    With .DayNameFont
      .Bold = GetSetting("Chapter6", "Startup", "DayNameFontBold", "False")
      .Charset = GetSetting("Chapter6", "Startup", "DayNameFontCharset", "77")
      .Italic = GetSetting("Chapter6", "Startup", "DayNameFontItalic", "False")
      .Name = GetSetting("Chapter6", "Startup", "DayNameFontName", "MS Sans Serif")
      .Size = GetSetting("Chapter6", "Startup", "DayNameFontSize", "10")
      .Strikethrough = GetSetting("Chapter6", "Startup", _
        "DayNameFontStrikethrough", "False")
      .Underline = GetSetting("Chapter6", "Startup", "DayNameFontUnderline", "False")
    End With

  End With

End Sub
```

When you start the program, everything will look as it has before. This is because the default settings we've added in the Load event are the same as the calendar's default settings. Select View | Fonts | Day Name Font from the form's menu bar, and the Font dialog will appear:



** *FontDlg.bmp* **

Select a new font from the dialog and click OK. The newly selected font will be transferred into the DayNameFont property and the calendar's appearance will be changed. Repeat the process with the DayFont property.

You may have noticed that changing the color of the selected font in the dialog has no effect on the font in the calendar.

> *The StdFont object makes no provision for encapsulation of the color of the selected font. If we want to still call the Font dialog via a function, extra code must be written to do so.*

This could easily be implemented by adding another parameter to be passed to and from the SelectFont function.

The System Tray Icon Control

The System Tray Icon Control, Systray.ocx, encapsulates everything necessary to place an icon in the system tray at any time. It has very few properties and even fewer events, but since the control itself is to be placed on a form, the control can set any properties or call any event for the form it's placed on. Just like the MSVBCalendar control, this control is supplied as source code , so the first thing we'll need to do is build it.

Building The Control

The source code for the System Tray Icon Control is located on the Visual Basic 5 CD in the `\tools\unsupprt\systray` directory. These files have their "read-only" property set to True, so we'll need to relocate them to the hard drive and change their properties.

Follow these steps to build the control:

1 Copy the files in the **`\tools\unsupprt\systray`** directory from the CD to a directory on the local hard drive. I used **`e:\vb\Projects\Systray`** for this particular project.

2 Highlight all of the files in the new directory. Right-click on the list and select "Properties" from the resulting pop-up menu.

3 Clear the "Read-only" check box on the General tab and click "OK".

4 From the Visual Basic 5 IDE, open the project file Systray.vbp. You'll notice that there are two files within the project, the module mSystray and the UserControl cSystray.

5 From the File menu, select "Make SysTray.ocx…". In the resulting "Make Project" dialog box, the selected directory will be the project directory. Change the directory in order to make the file as **`c:\windows\system\SysTray.ocx`** and click "OK".

The control will then be listed as in the Components dialog as "System Tray Icon Control". Select this item in the Chapter6 project so we can use it.

Properties

As I said, the System Tray Icon Control has very few properties (and no functions):

Property	Description
Name	Name given to the control
Index	Returns/sets the number identifying the control in a control array
Left	Returns/sets the distance between the left edge of the control and its container
Tag	Stores any extra data
Top	Returns/sets the distance between the top edge of the control and its container
InTray	Whether or not the icon is currently visible in the tray
TrayIcon	Returns/sets the icon used with the control
TrayTip	Returns/sets the control's tooltip

Events

There are only four events for the control itself, all related to the mouse:

Event	Description
MouseDblClick	Occurs when a mouse button is double-clicked
MouseDown	Occurs at the end of the down-stroke of a mouse button
MouseMove	Occurs when the mouse is moved over the icon in the system tray
MouseUp	Occurs at the end of the up-stroke of a mouse button

However, since the three events other than MouseMove also identify the mouse button that was clicked, that makes for seven individual events … all from single system tray icon!

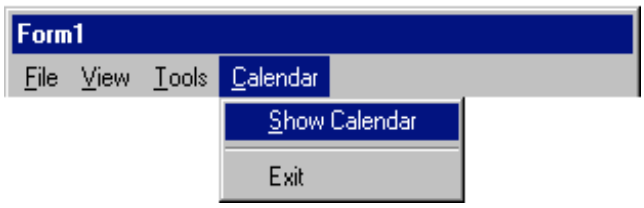
Try It Out – The System Tray Icon Control

Setting-Up The Form

Place the System Tray Icon Control onto our form in the Chapter6 project:

*** SysTray.bmp ***

The first thing we need to do is set the form's visible property to False, so the icon shows up in the system tray, ready to make the form appear. Place a menu named Calendar, with the handle of mnuCalendar, onto the form. Give this menu three items: Show Calendar (mnuShowCalendar), mnuBar1, and Exit (mnuExit):



*** SysMenu.bmp ***

Now, go back into the menu editor and set the Visible property for mnuCalendar to False. Doing so hides this menu, making it a candidate for a popup menu.

The Show Calendar menu item will be used for causing the calendar form itself to be displayed. Place the following code under the mnuShowCalendar_Click event:

```
Private Sub mnuShowCalendar_Click()

  ' Show the Calendar form
  Form1.Visible = True

End Sub
```

This next piece of code goes under the mnuExit_Click event, to do exactly what it says:

```
Private Sub mnuExit_Click()

  ' Exit the program
  End
```

```
End Sub
```

Back on the File menu, this is the code for the mnuFileClose_Click event. This causes the form to disappear, making it available for later. Notice this doesn't cause the program to end, so the icon will still be visible in the system tray:

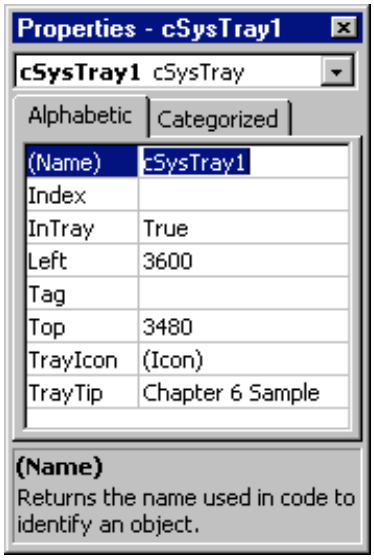
```
Private Sub mnuFileClose_Click()

  ' Make the form invisible
  Form1.Visible = False

End Sub
```

And lastly, this code goes under the control's MouseDown event. It causes the Calendar menu to pop up over the icon on the left-click event:

```
Private Sub cSysTray1_MouseDown(Button As Integer, Id As Long)

  Select Case Button
    Case 1   ' Left button
      PopupMenu mnuCalendar
    Case 2   'Right button
      ' Do nothing
  End Select

End Sub
```

Setting the Control's Properties

The control's properties are set as shown:



*** STProp.bmp ***

The icon is in the tray by default when the setting for the InTray property is True. The icon itself is set as needed (I used trffc10a.ico in the **VB\Graphics\Icons\Traffic** directory to get the traffic light icon), and the TrayTip property is loaded with the string we want to be displayed when the mouse hovers over the icon, "Chapter 6 Sample".

Running the Program

Start the program from the Run menu in the VB IDE and the icon will appear in the system tray. Notice that the form does not appear. Hover the mouse over the icon and the tray tip will appear:

*** TrayTip.bmp ***

Clicking the left mouse button on the icon will cause the popup menu to appear:

*** TrayMenu.bmp ***

Clicking on Show Calendar will show the form as we've been using it all along. Use the File | Close menu item to close the form, leaving the icon active in the system tray. Clicking on the Exit menu item on the popup menu will carry out the End command and close our program.

System Tray Icon Control GUI Notes

The System Tray Icon is a great addition to the GUI environment. It's small, out of the way, doesn't do anything until you need it to, and it's always there. But be careful. Too much of a good thing will always ruin it. On a monitor set for 640x480 pixels, I've found confusion after about 4 icons in the system tray. Larger resolutions (600 x 800, 1024x768, or even higher) can support more without much more confusion, but don't overdo it.

Summary

At this point you should be ready to tackle more of the unsupported controls and utilities provided on the Visual Basic CD. In this paper we've covered:

A calendar control with extensive capabilities that we can change for our own custom uses

A better set of methods for implementing most of the common dialogs

A control specifically used for implementing icons in the system tray