by Jonathan Wood

HIDE THE SLUDGE

A STARTUP SPLASH SCREEN CAN GIVE YOUR APP A FAST, PROFESSIONAL FEEL.

Many commercial applications display a startup splash screen as the application is loading. A splash screen offers two primary benefits: it's a place to display your program's copyright notice (and a flashy logo if you so desire) each time your program starts. It can also make your program seem more responsive because it is thrown onscreen quickly and remains there while the rest of the program loads, giving the user something to look at other than a blank display (see Figure 1).

In showing how to create a splash screen, I planned to demonstrate what I thought was the best possible technique. Unfortunately, it soon became apparent that no single technique was the best solution. Rather, a number of different problems must be solved and the best way to solve them depends on the nature of your application. I'll present several methods and point out the advantages and disadvantages of each one.

The simplest way to create a splash screen is to show a modal splash form and then wait until the splash form closes before continuing with the rest of your program's initialization. By selecting the Project command from the Options menu, you can set the Start Up Form to Sub Main and then show the splash screen from Sub Main before any other forms have been loaded:

```
Sub Main ()

   'Load splash screen (unloads automatically)
   frmSplash.Show 1

   'Load program form(s)
   frmMain.Show

End Sub
```

The "1" that follows the Show method causes the splash form (frmSplash) to be displayed modally. The result is that the Show method won't return until the splash form is unloaded. To unload the splash form, I add two routines to the splash form to set a timer for five-second intervals, and then unload the form when it receives a timer event:

```
Sub Form_Load ()

   'Fire up timer (5 second interval)
   Timer1.Interval = 5000

End Sub
```

```
Sub Timer1_Timer ()

   'Interval expired, unload form
   Unload Me

End Sub
```

The only problem with this simple technique is that if your program takes a long time to load, it will now take much longer. Visual Basic forms that have many controls can be slow to load. It would be nice if the program could continue loading while the startup screen is displayed. My second method displays the splash screen as a modeless form by loading it with this statement:

```
   frmSplash.Show
```

This causes the Show method to return immediately so initialization can continue while the splash form is displayed. However, this approach requires that you address some additional issues. To begin with, you need to make the splash screen a topmost window so that it stays on top of any other forms that get loaded during initialization (see Listing 1).

You also need to prevent the user from accessing the main form while the splash screen is displayed. To do this, the main form's Load event sets its Enabled property to False. When the splash form unloads, its Unload event sets the main form's Enabled property back to True. If your application has more than one top-level form, you may need to add code to disable and then re-enable additional forms.

If your program triggers an error that results in a message box being displayed during initialization, the program becomes locked. This is because message boxes are modal and prevent the

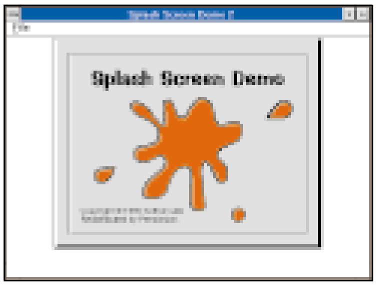

FIGURE 1 ***Splashy Screen.*** *Use a splash screen to give your users a nice picture to look at while your program is loading. You can also use a splash screen to hide forms as they are being populated with data.*

Jonathan Wood writes commercial and custom software in Visual Basic, Visual C++, and assembly language. His company, SoftCircuits, is located in Irvine, California. Reach him on the VBPJ *Forum (he's the section leader of the DLL/API Lab section) and* MSBASIC *Forum on CompuServe at 72134,263.*

splash screen from getting the timer event it needs to unload itself. Because the splash screen is now a topmost window, it will cover the message box and the user won't be able to dismiss it. Check carefully for any errors that may occur while the splash form is displayed. If you find potential errors, you must trap them and make sure the splash form is unloaded before displaying any error messages.

My last method involves the use of a small C program that displays the splash screen. I wrote such a program called STARTUP.EXE that I invoke from Sub Main:

```
Sub Main ()
   Dim hInst As Integer

   'Run splash program (unloads automatically)
   hInst = Shell(App.Path & "\STARTUP.EXE")

   'Load program form(s)
   frmMain.Show

End Sub
```

The STARTUP.EXE program simply displays a bitmap in a topmost window and automatically closes it after five seconds. If you have a resource editor, you can change the bitmap displayed by this program.

Using this method, the splash screen is now part of another program so the problem of an error message preventing the splash screen from unloading is gone. Whereas before you re-enabled the main form when the splash screen unloaded, you are no longer notified when this occurs. There is no simple workaround for this.

Perhaps the primary reason for using a small C program this way is to have the splash screen load faster than is possible from a large Visual Basic program. However, there really isn't much of a speed advantage here. The program is not executed until Sub Main can invoke it anyway, which makes it about the same speed as my second method. If time is critical, you might make a small C program that displays the splash screen and then invokes your primary Visual Basic program. This way, the splash screen is displayed almost instantly.

The source code for the three splash screen demos and the startup C program are available in the *VBPJ* and MSBASIC Forums on CompuServe as SPLASH.ZIP (they will be available on the third volume of the VB-CD Quarterly as well). As I pointed out at the start, there is no best way to display a splash screen in Visual Basic, but it can be done. Hopefully one of the three methods I've demonstrated, or perhaps even a combination of them, will be just what your application needs to give it that something extra.

```
'Windows declarations
Declare Sub SetWindowPos Lib "User" (ByVal hWnd _
  As Integer, ByVal hWndInsertAfter As Integer, _
  ByVal X As Integer, ByVal Y As Integer, _
  ByVal cx As Integer, ByVal cy As Integer, _
  ByVal wFlags As Integer)

'Windows constants
Global Const HWND_TOPMOST = -1
Global Const SWP_NOMOVE = &H2
Global Const SWP_NOSIZE = &H1

Sub Form_Load ()

  'Make form a top-most window
  Call SetWindowPos(hwnd, HWND_TOPMOST, 0, 0, _
     0, 0, SWP_NOMOVE Or SWP_NOSIZE)

  'Force painting of form
  Visible = True: Refresh

  'Fire up timer (5 second interval)
  Timer1.Interval = 5000

End Sub
```

LISTING 1 ***From the Top.*** *This form's Load event makes the splash screen a topmost window by calling the Windows SetWindowPos API function. Notice the statements that force the form to be painted. Because the program initialization proceeds immediately, the splash form won't get another chance to display itself until the program completes initialization and becomes idle.*

API TECHNIQUE: ECHO OFF FOR LIST BOXES

If you've ever had to fill a list box with many items, or programmatically select all items in a multiselect list box, you probably noticed flickering as the list box was updated: as each item is modified, Windows repaints the list box to reflect the changes.

In such cases, it makes sense to prevent the list box from updating until all the changes are complete: an "echo off" for list boxes. Not only would this prevent the list box from flickering as it is updated, but it would also make it considerably faster because the list box would only be repainted once.

Fortunately, the Windows API provides this functionality through the WM_SETREDRAW message. This message sets or clears the redraw flag for a given window. While the WM_SETREDRAW message may be sent to any window, it makes sense to use it only for certain window types.

To demonstrate the WM_SETREDRAW message, the example code selects all of the items in a multiselect list box (see Listing 2).

The function uses the Windows API function SendMessage to send the WM_SETREDRAW message to the list box with an argument of False. This clears the redraw flag for the list box while all the list items are selected.

The WM_SETREDRAW message is then sent again with an argument of True to set the redraw flag back on. While this is usually all you need, the Windows Software Development Kit documentation recommends that you then call InvalidateRect with the fErase parameter set to True. This instructs the list box to repaint itself entirely to prevent any garbage from being left behind, which can occur as a result of some list box operations.

KEYWORD OF THE MONTH: DIR$

The Dir$ function returns the names of files that match a specified pattern and attribute. For example, this statement prints the name of the first file in the current directory:

```
   Print Dir$("*.*")
```

To get additional files matching the same pattern, call Dir$ with no arguments. When no more files remain, Dir$ returns "". To print all the files in the current directory, you could use:

```
Sub Command1_Click ()
   Dim fName As String

   fName = Dir$("*.*")

   Do Until fName = ""
      Print fName
      fName = Dir$
   Loop

End Sub
```

Note that Dir$ returns files in the order of DOS's internal directory tables. You'll have to process the list if you need it sorted.

The flexibility of the Dir$ function makes it useful for other tasks as well. For example, this function checks to see if a specified file exists or not:

```
Function FileExists (fName As String)

   On Error Resume Next

   If Dir$(fName) = "" Then
      FileExists = False
   Else
      FileExists = True
   End If

End Function
```

If the file specified by fName exists, the function returns True; otherwise, it returns False. The fName argument can be a simple file name or it can be a complete, fully qualified file name with drive and path specifier. Notice that I used an On Error Resume Next statement before calling Dir$. This is required in case the file name is specified as existing on a floppy drive. If there is an error reading the drive (if the drive is empty, for example) then the function returns False instead of triggering an error.

You can also have the Dir$ function return files that have certain file attributes. This function returns a drive's volume label:

```
Global Const ATTR_VOLUME = &H8

Function GetVolume (DriveSpec As String) As String
   Dim i As Integer, VolName As String

   On Error Resume Next

   VolName = Dir$(DriveSpec, ATTR_VOLUME)

   'Remove "." returned by Dir$
   i = InStr(VolName, ".")
   If i Then
      GetVolume = Left$(VolName, i - 1) & _
         Mid$(VolName, i + 1)
   Else
      GetVolume = VolName
   End If

End Function
```

The Dir$ function can return the volume label because DOS stores volume labels as a special type of file. However, since Dir$ thinks it's returning a file name, a period is inserted into the volume name. The GetVolume function uses InStr to find the period and remove it. The argument is the drive for which you want the volume label returned. For example, to print the volume label for drive C:, you would use:

```
   Print GetVolume("C:")
```

```
Global Const WM_SETREDRAW = &HB

Declare Function SendMessage Lib "User" (ByVal _
  hWnd As Integer, ByVal wMsg As Integer, _
  ByVal wParam As Integer, lParam As Any) _
  As Long
Declare Sub InvalidateRect Lib "User" (ByVal _
  hWnd As Integer, ByVal lpRect As Long, _
  ByVal bErase As Integer)

Sub Command1_Click ()
  Dim i As Integer

  'Disable updates
  i = SendMessage(List1.hWnd, WM_SETREDRAW, _
     False, ByVal 0&)

  'Select all items in list
  'MultiSelect property must be 1 or 2
  For i = 0 To List1.ListCount - 1
     List1.Selected(i) = True
  Next i

  'Reenable updates
  i = SendMessage(List1.hWnd, WM_SETREDRAW, _
     True, ByVal 0&)

  'Force update of entire list box
  Call InvalidateRect(List1.hWnd, 0&, True)

End Sub
```

LISTING 2 ***Suppress List Box from Redrawing.*** *This function uses the Windows API function SendMessage to keep a list box from redrawing, which makes the list box look more professional and speeds it up, too.*

DISTRIBUTION TIP: DIRECTORY FOR VBX/DLL FILES

For all the convenience that custom controls bring to Visual Basic, VBXs cause most of the conflicts that occur when you distribute applications. Such applications are likely to find themselves on systems with other applications created with Visual Basic possibly using the same VBXs.

For this to work without conflict, it's important that Visual Basic developers adhere to a common set of conventions. Primarily, VBXs and DLLs that might be shared with other applications should be placed in the Windows system directory. Visual Basic's Setup Wizard handles this automatically for you. Unfortunately, some programs do not follow this protocol and install VBX files in the Windows directory.

Maybe half of the support calls I handle concern users who try to run my program, but are thwarted by an error message stating that a particular VBX used by the program is out of date. How can the VBX be out of date when they just used my installation program, which installs my version of the VBX? Invariably, the answer is that another program installed an older version of the VBX in the Windows directory.

When Windows loads a DLL (VBX files are a type of DLL), it searches for it in these places:

- Modules already loaded into memory.
- The active directory.
- The Windows directory.
- The Windows system directory.
- The directory where the client program resides (Windows 3.1 and later).
- Directories listed in the PATH environment string.
- Directories mapped in a network.

When my program requests the VBX, Windows locates the version in the Windows directory before it finds my version in the Windows system directory. If the one in the Windows directory is older than the one my program uses, the error occurs. If I tell the user to delete or rename the VBX in the Windows directory, my program runs fine.

What happens if the other program had correctly installed the VBX in the Windows system directory? Setup programs created with Setup Wizard compare the internal version information of VBX files. If the existing version is newer than the one being installed, the existing file is not overwritten. In this case, the existing VBX was an older version, so the newer one would have been installed on top of it. Because newer versions of VBXs should always be compatible with older versions, this allows both programs to peacefully coexist. ■