Visual Studio Magazine Online
Classic VB Corner

Discarding Dependencies

Why tempt fate by making your application dependent upon an easily replaced component? Karl Peterson shows how you can begin reducing external dependencies in your VB apps.

December 14, 2010 · by Karl E. Peterson

The world's gone mad, it seems sometimes, with security concerns. Every time you turn on the radio, look at a newspaper, or can't avoid listening to a politician, you will to read or hear all about all the latest threat(s) you really ought to be scared of.

If you're not yet thoroughly paranoid, consider the wild notion that Microsoft could, if they were so inclined, slowly erode the base of existing Classic VB applications out there by simply breaking components they rely upon. Think that's crazy? I would suggest that given an appropriate "security concern" they would likely not hesitate to act. Search for killbits to see this theory in action in only a slightly different arena.

I'd like to suggest that there are a good number of easy steps you can take, to begin reducing external dependencies. One of the simpler ones to drop is the common dialogs ActiveX control. This control, comdlg32.ocx, is simply a wrapper around a handful of API function calls. Nothing fancy at all, and yet it's used as a lazy convenience by many.

Over a decade ago, Bruce McKinney wrote the classic *Hardcore Visual Basic*, which Microsoft later granted permission to be republished on my website (when they removed it from theirs). Bruce had a magical way with words. Not sure I have ever read another programming book quite like that one. I will always wish I had his gift there.

But we had a fundamental disagreement on the way all his code was so, well, dependent on the rest of it. I guess this feeling was mainly based on Bruce's use of typelibs for everything. In theory, typelibs could free the developer from the drudgery of declares, but in practice it often meant more work when it came time to reuse code.

So what I've done over the years was gradually, as I needed them, taken his common dialog routines and extracted them from their dependencies. This final touch-up not only freed me from comdlg32.ocx, but from any external at all. I can now drop the desired module(s) directly into any project, and begin using the dialogs right away.

I'll be reposting these over the next few months on my site, as well as writing a bit about them here. Taking the API calls outside the OCX not only offers total flexibility in calling them, but in doing so it offers new capabilities with some of them.

So let's start with the ChooseFont dialog, since I brought up its new behavior in my last column, and it's really one of the simplest. The declare is straight-forward:

```
Private Declare Function ChooseFont Lib "comdlg32.dll" _
   Alias "ChooseFontA" (pChoosefont As ChooseFont) As Long
```

To call ChooseFont, you just fill up a few elements of the following structure:

```
Private Type ChooseFont
   lStructSize As Long         ' Filled with UDT size
   hWndOwner As Long           ' Caller's window handle
   hDC As Long                 ' Printer DC/IC or NULL
   lpLogFont As Long           ' Pointer to LOGFONT
   iPointSize As Long          ' 10 * size in points of font
   Flags As Long               ' Type flags
   rgbColors As Long           ' Returned text color
   lCustData As Long           ' Data passed to hook function
   lpfnHook As Long            ' Pointer to hook function
   lpTemplateName As String    ' Custom template name
   hInstance As Long           ' Instance handle for template
   lpszStyle As String         ' Return style field
   nFontType As Integer        ' Font type bits
   iAlign As Integer           ' Filler
   nSizeMin As Long            ' Minimum point size allowed
   nSizeMax As Long            ' Maximum point size allowed
End Type
```

Bruce wrapped this into a fairly tight little routine, which I've only embellished a bit. The idea here is kind of cool. You call GetFontChoice by passing a Font object, which is used as the starting point for initializing the dialog. This makes it very simple to change the font in an existing control by simply passing its .Font property as this single required parameter. The remaining optional parameters just fine tune how you want the dialog to behave.

```
Public Function GetFontChoice(InitFont As Font, _
   Optional ByVal PrinterDC As Long = -1, _
   Optional ByVal hWnd As Long = -1, _
   Optional Color As Long = vbBlack, _
   Optional ByVal MinSize As Long = 0, _
   Optional ByVal MaxSize As Long = 0, _
   Optional Flags As ChooseFontFlags = 0) As Boolean

   Dim cf As ChooseFont
   Dim fnt As LogFont
   Dim fntname() As Byte

   ' Since we allow any flags to be passed in, we need to have
   ' a way to turn off flags not supported by this routine.
   ' (Some exercise left to the reader!)
   Const UnwantedFlags As Long = _
      CF_APPLY Or CF_ENABLEHOOK Or CF_ENABLETEMPLATE

   ' Make the code a bit more readable.
   Const PointsPerTwip As Long = 1440 / 72

   ' Flags can get reference variable or constant with bit flags.
   ' User may specify printer fonts with flag or hDC.
```

```
If PrinterDC = -1 Then
   PrinterDC = 0
   If Flags And CF_PRINTERFONTS Then PrinterDC = Printer.hDC
Else
   Flags = Flags Or CF_PRINTERFONTS
End If

' Must have screen, printer, or both!
If (Flags And CF_PRINTERFONTS) = 0 Then
   Flags = Flags Or CF_SCREENFONTS
End If

' MinSize can be minimum size accepted
If MinSize Then Flags = Flags Or CF_LIMITSIZE
' MaxSize can be maximum size accepted
If MaxSize Then Flags = Flags Or CF_LIMITSIZE

' Put in required internal flags and remove unsupported
Flags = (Flags Or CF_INITTOLOGFONTSTRUCT) And Not UnwantedFlags

' Initialize LOGFONT variable
With InitFont
   fnt.lfHeight = _
      -(.Size * (PointsPerTwip / Screen.TwipsPerPixelY))
   fnt.lfWeight = .Weight
   fnt.lfItalic = .Italic
   fnt.lfUnderline = .Underline
   fnt.lfStrikeOut = .Strikethrough
   fntname = StrConv(.Name & vbNullChar, vbFromUnicode)
   Call CopyMemory(fnt.lfFaceName(0), fntname(0), Len(.Name) + 1)
   ' Other fields zero
End With

' Initialize CHOOSEFONT variable
cf.lStructSize = Len(cf)
If hWnd <> -1 Then cf.hWndOwner = hWnd
cf.hDC = PrinterDC
cf.lpLogFont = VarPtr(fnt)
cf.iPointSize = InitFont.Size * 10
cf.Flags = Flags
cf.rgbColors = Color
cf.nSizeMin = MinSize
cf.nSizeMax = MaxSize
' All other fields zero

If ChooseFont(cf) Then
   GetFontChoice = True
   Flags = cf.Flags
   Color = cf.rgbColors
   With InitFont
      .Bold = cf.nFontType And BOLD_FONTTYPE
      '.Italic = cf.nFontType And ITALIC_FONTTYPE
      .Italic = fnt.lfItalic
      .Strikethrough = fnt.lfStrikeOut
      .Underline = fnt.lfUnderline
      .Weight = fnt.lfWeight
      .Size = cf.iPointSize / 10
      .Name = TrimNull(StrConv(fnt.lfFaceName, vbUnicode))
   End With
Else
```

```
      ' CommDlgExtendedError returns 0 for Cancel, <>0 for errors.
      Debug.Print _
         "CommDlgExtendedError = &h"; Hex$(CommDlgExtendedError())
      GetFontChoice = False
   End If
End Function
```

The first third of the GetFontChoice function merely twiddles flags, based on what was passed in the optional parameters. The CHOOSEFONT structure that we pass to the ChooseFont function contains a pointer to a LOGFONT structure that's used to initialize the dialog selections for things like facename, size, bold, italic and so on.

A bit of gymnastics is required to properly build these structures in the middle section of the routine, because we need to do the Unimess (Unicode-to-ANSI) conversion ourselves, since we're passing a pointer to one structure from within another. But it all comes together and finally we make the call to ChooseFont. A successful return means we just reassign the chosen attributes to the InitFont object, and our client is off and running. It sounds like a lot of work, but to actually use this in an application is as simple as dropping the module in and calling it like this:

```
   Dim Color As Long
   Dim Flags As ChooseFontFlags
   ' Simplest possible way to call the ChooseFont API,
   ' by wrapping it up in a single routine and passing
   ' the font object we want to change.
   With Label1
      ' Set the flags we want to use.
      Flags = CF_SCREENFONTS Or CF_NOVERTFONTS _
            Or CF_EFFECTS Or CF_SCALABLEONLY
      ' Only the color needs special handling, since that's
      ' an attribute of the label and not the font.
      Color = .ForeColor
      If GetFontChoice(.Font, , Me.hWnd, Color, , , Flags) Then
         ' Success!
         .ForeColor = Color
      End If
   End With
```

Here I used a Label control to demonstrate, by passing its .Font object for the GetFontChoice function to update with a call to ChooseFont. Note that the only "attribute" of the .Font object that needs special handling is its color, because that's actually an attribute of the control not the font. If "any color you'd like, as long as it's black" will do, the whole call can be reduced to a one-liner.

I hope to get all my common dialog wrapper updates out on the web soon. Please see the Dialogs sample on my site, and watch it for updates as new columns come out here.

About the Author

Karl E. Peterson wrote Q&A, Programming Techniques, and various other columns for VBPJ and VSM from 1995 onward, until Classic VB columns were dropped entirely in favor of other languages. Similarly, Karl was a Microsoft BASIC MVP from 1994 through 2005, until such community contributions were no longer deemed valuable. He is the author of VisualStudioMagazine.com's new Classic VB Corner column. You can contact him through his Web site if you'd like to suggest future topics for this column.

1105 Redmond Media Group
Copyright 1996-2010 1105 Media, Inc. View our Privacy Policy.