Unicode Strings You Along

Click & Retrieve Source CODE!

Bypass automatic conversion of strings to avoid corruption.

by Karl E. Peterson

VB4/32 maintains all strings internally as Unicode, but always converts them to ANSI before passing them to external routines. When ANSI strings are returned to VB, VB converts the strings to Unicode. This process not only increases the overhead of passing strings, but also might corrupt strings if the operating system doesn't map both conversions in the same way. In a previous column [Programming Techniques, "Chef's Surprise," *VBPJ* May 1996], I presented a function that could detect such corruption. Now I'll show you how to bypass these automatic conversions.

Windows NT introduced the Unicode character set to PCs to overcome ASCII's (or ANSI's) limitation of 256 characters. By doubling the number of bits allotted to each character from eight to 16, Unicode offers the opportunity to store 65,536 unique characters within any given font or code page—enough for every character from every language ever used on Earth, legend has it. This would indeed seem to be a good thing. Unfortunately, Unicode presents unique challenges to the VB programmer, as VB4 seems designed specifically to prevent you from taking advantage of it. I will discuss these problems a little later.

Nearly all Win32 API functions that accept strings as parameters (even if the strings are passed within a user-defined type) have two versions. Typically known as the "A" and "W" versions, for ANSI and Wide (Unicode), these two variations are differentiated by one of these two letters appended to the function name. The "A" functions expect to receive ANSI strings, and the "W" functions expect to receive Unicode strings—to send the wrong type invites almost certain failure. To preserve consistency with existing code, many developers alias the "A" functions to their Win16 names:

```
Private Declare Function SendMessage _
   Lib "user32" Alias "SendMessageA" _
   (ByVal hwnd As Long, ByVal wMsg As Long, ByVal _
   wParam As Long, lParam As Any) As Long
```

Normally, if you pass a string in lParam, VB4 will convert it to ANSI and the SendMessage function will work just fine (as long as the string is preceded with the ByVal keyword in the call). But what if you want to pass a Unicode string in lParam? You will need an additional declaration to specifically reference the "W" version of SendMessage:

```
Private Declare Function SendMessageB _
   Lib "user32" Alias "SendMessageW" _
   (ByVal hwnd As Long, ByVal wMsg _
   As Long, ByVal wParam As Long, _
   lParam As Any) As Long
```

The only differences between these two declarations are the suffix on the internal name of the function and the suffix I've added to the function name as VB will know it. I chose to use "B" rather than "W" because you must convert Unicode strings into a byte array before you can pass them intact from VB.

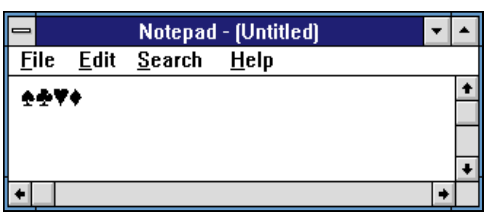


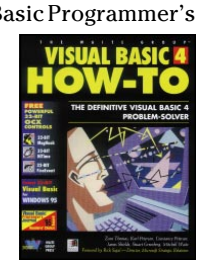
FIGURE 1 ***Unicode Support In Notepad.*** *NT's native controls support Unicode by default. Even though Notepad may appear to be using a symbol font such as WingDings, it's actually using Times New Roman. The Visual Basic program sent these characters to Notepad with SendMessageW.*

Not all the fonts that ship with Windows NT are the standard ANSI fonts we're all familiar with; instead, NT ships with an assortment of Unicode fonts. While these fonts are not rich in characters, they contain a number of extended character sets that you can browse using NT's Character Map applet. If you select these extended characters into the clipboard while using Character Map, you can paste them into Unicode-enabled controls, among which are the NT-provided standard controls such as edit boxes. For example, the Notepad applet, which ships with NT, readily accepts Unicode characters when pasted from the clipboard or read from a file (see Figure 1).

This provides a handy method to test what's required to pass Unicode strings from VB. Use the ChrW function to build strings containing these extended characters, and convert them by assigning the string to a dynamic byte array. Representing the four card suits (and a terminating null), these extended character codes were determined using the Character Map applet:

```
Dim msg As String
Dim msgB() As Byte
msg = ChrW(&H2660) & ChrW(&H2663) _
```

Karl E. Peterson is a GIS Analyst with a regional transportation planning agency and a member of the Visual Basic Programmer's Journal *Technical Review Board. Based in Vancouver, Washington, he's also an independent programming consultant and a writer. Karl coauthored* Visual Basic 4 How-To*, from Waite Group Press. Online, he's a section leader in the* VBPJ *Forum 32-Bit Bucket section and a Microsoft Developer Support MVP. Contact Karl on CompuServe at 72302,3707.*



```
   & ChrW(&H2665) & ChrW(&H2666) & ChrW(0)
msgB = msg
```

Though not all Unicode fonts may have characters in these positions, if they do, they should always be these specific characters. That's really the beauty of Unicode.

If you pass the string as is, using the "A" version of SendMessage, VB converts it to ANSI before passing it to SendMessage. Because the extended Unicode character codes are outside the range of the ANSI character set, the resultant string consists of four question marks. To pass the string without the default corruption occurring, you must use the "W" version of SendMessage. The next example first spawns an instance of Notepad, then determines the handles for both the main window and the text box within it (which conveniently happens to be the first child of the main window, allowing use of the GetWindow API). A pointer to the byte array is sent to Notepad's text box using the "W" version of SendMessage by passing a reference to the first element of the byte array. If you run this example in NT and see four boxes rather than the four card suits in Notepad, you need to change the font Notepad is using to one that supports Unicode, such as Times New Roman (see Figure 1):

```
Dim hWndNotepad As Long
Dim hWndNotepadText As Long
Shell "Notepad", vbNormalFocus
hWndNotepad = _
   FindWindow(vbNullString, _
   "Notepad - (Untitled)")
hWndNotepadText = _
   GetWindow(hWndNotepad, GW_CHILD)
Call SendMessageB(hWndNotepadText, _
   WM_SETTEXT, Len(msg), msgB(0))
```

Although Unicode offers some exciting possibilities, you can encounter some interesting challenges when you attempt to make use of this expanded functionality. Windows 95's general lack of support for Unicode only compounds these challenges. Other than to acknowledge it exists, Windows 95 returns a failure code on most calls that attempt to use Unicode functions. Under NT, VB4/32 can call Unicode functions and pass Unicode "strings" to external programs that understand them, but unfortunately I've found no way to actually display Unicode strings within a VB application.

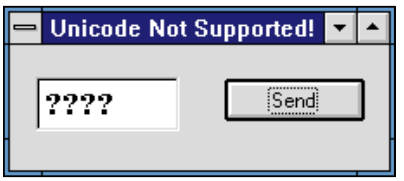


FIGURE 2 ***No Unicode Support In Visual Basic.*** *Apparently for compatability with Windows 95, Microsoft decided to explicitly disable NT's native Unicode support. If a true Unicode string is sent to an intrinsic VB control, the control displays it as a series of question marks, regardless of which font is selected into the control.*

You can't paste one into a text box from the clipboard, and you can't directly assign one to the Text property of a text box. No matter what you do, you are left with those four meaningless question marks signifying that there was no appropriate character available in the Unicode-to-ANSI conversion that always occurs (see Figure 2). Even using the "W" version of SendMessage

VB4

```
Option Explicit
'
' Win32 APIs to determine OS information.
'
Private Declare Function GetVersionEx Lib "kernel32" _
  Alias "GetVersionExA" (lpVersionInformation As _
  OSVERSIONINFO) As Long
Private Type OSVERSIONINFO
  dwOSVersionInfoSize As Long
  dwMajorVersion As Long
  dwMinorVersion As Long
  dwBuildNumber As Long
  dwPlatformId As Long
  szCSDVersion As String * 128
End Type
Private Const VER_PLATFORM_WIN32s = 0
Private Const VER_PLATFORM_WIN32_WINDOWS = 1
Private Const VER_PLATFORM_WIN32_NT = 2
'
' Member variables.
'
Private m_os As OSVERSIONINFO
Private m_NT As Boolean
Private m_95 As Boolean

' ================================
'  Initialize
' ================================
Private Sub Class_Initialize()
  '
  ' Retrieve version data for OS.
  '
  m_os.dwOSVersionInfoSize = Len(m_os)
  Call GetVersionEx(m_os)
  '
  ' Determine and store values likely to be
  ' referenced often.  VB4/32 will not run
  ' in Win32s, so needn't test for that.
  '
  Select Case m_os.dwPlatformId
     Case VER_PLATFORM_WIN32_WINDOWS
        m_95 = True
        m_NT = False
     Case VER_PLATFORM_WIN32_NT
        m_95 = False
        m_NT = True
  End Select
End Sub

' ================================
'  Public Properties
' ================================
Public Property Get MajorVersion() As Long
  MajorVersion = m_os.dwMajorVersion
End Property

Public Property Get MinorVersion() As Long
  MinorVersion = m_os.dwMinorVersion
End Property

Public Property Get BuildNumber() As Long
  BuildNumber = WordLo(m_os.dwBuildNumber)
End Property

Public Property Get PlatformID() As Long
  PlatformID = m_os.dwPlatformId
End Property
```

CONTINUED ON PAGE 155.

LISTING 1 ***What OS Am I In?*** *This class module, COpSysInfo, returns a number of useful properties identifying the operating system that your application is running under. It's most useful for determining which code block to call when the requirements of Windows NT and Windows 95 differ.*

VB4

CONTINUED FROM PAGE 154.

```
Public Property Get IsWinNT() As Boolean
  IsWinNT = m_NT
End Property

Public Property Get IsWin95() As Boolean
  IsWin95 = m_95
End Property

Public Property Get Platform() As String
  If m_95 Then
     Platform = "Windows 95"
  Else 'm_NT
     Platform = "Windows NT"
  End If
End Property

Public Property Get Version() As String
  '
  ' Build and return version info string.
  '
  Version = Platform & _
   " v" & MajorVersion & _
   "." & MinorVersion & _
   ", Build " & BuildNumber
End Property

Public Property Get CSDVersion() As String
  CSDVersion = m_os.szCSDVersion
End Property

' ================================
'  Private Methods
' ================================
Private Function WordLo(LongIn As Long) As Integer
  '
  ' Low word retrieved by masking off high word.
  ' If low word is too large, twiddle sign bit.
  '
  If (LongIn And &HFFFF&) > &H7FFF Then
     WordLo = (LongIn And &HFFFF&) - &H10000
  Else
     WordLo = LongIn And &HFFFF&
  End If
End Function
```

doesn't work with VB controls:

```
Call SendMessageB(Text1.hwnd, _
   WM_SETTEXT, Len(msg), msgB(0))
```

VB4's lack of support for Unicode presumably stems from Microsoft developers wanting to maintain compatibility with Windows 95, which has virtually no built-in support for Unicode at all. With very few exceptions (see Knowledge Base article Q125671), Windows 95 implements the "W" functions as stubs, and these functions will return failure every time. Also with very few exceptions, nearly every Win32 API in NT that uses string parameters supports both the "A" and "W" versions. The most glaring exceptions are 70 or so of the NetXXX functions, which support only Unicode strings.

WHICH WIN32 IS THIS?

Microsoft has consistently downplayed the Win32 API differences between Windows NT and Windows 95, but if you've ever needed to support one app in both environments and used more than a handful of API calls in it, you know that the differences can range from subtle to staggering. The nonsupport of Unicode in Windows 95 makes for a prime example, and I'm sure you've all run into others that just drove you berserk as you fought to find workarounds. VB4 provides the Win16 and Win32 conditional compilation constants, but doesn't help you determine *which flavor* of Win32 you're actually running under.

I've put together a class module you can use in VB4/32 to decide how to proceed based on the operating system in use. (For the same capabilities using 16-bit VB3 or VB4, see Programming Techniques, "Creeping Version-itis," in the July 1995 issue of *VBPJ*.) You can either create instances of it as needed or create one global instance and refer to it whenever you must decide which code to execute based on the operating system you find yourself in.

As I've pointed out in previous columns, many Win32 API calls have been greatly enhanced over their Win16 counterparts. The GetVersionEx API function definitely marks a vast

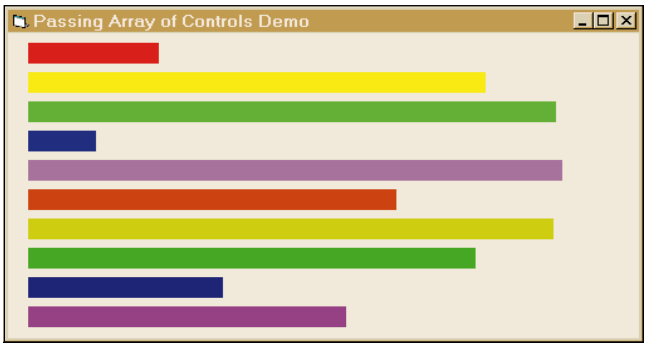


FIGURE 3 ***Makeshift Graph Control.*** *One way to create a bar chart is to simply use a control array of labels with varying colors and widths. This technique can take less time to construct and require fewer resources than using a VBX or OCX, if it suits your needs.*

improvement over the older GetVersion API function. The Initialize event of the COpSysInfo class makes a call to GetVersionEx and stores the returned OSVERSIONINFO data structure so that you can query it as needed (see Listing 1).

This OSVERSIONINFO structure contains much more useful data than that returned by GetVersion. One element indicates whether you're running under Win32s, Windows (at this time, that only applies to Windows 95, but you should consider it as "Win32, but not NT"), or Windows NT. The Initialization event also sets the value of two Boolean member variables used to indicate whether Win95 or NT is running because you will probably require this information more than anything else. (VB4/32 applications will not run under Win32s). If you have a section of code that runs only in NT, you can decide whether to proceed like this:

```
' Don't bother if not running NT.
'
Dim os As New COpSysInfo
If os.IsWin95 Then
   MsgBox "Win95 doesn't support Unicode", _
      vbInformation, "Cannot Proceed"
   Exit Sub
End If
```

The COpSysInfo class includes numeric properties that return the MajorVersion, MinorVersion, and Build, all of which are simply read from the OSVERSIONINFO structure. One problem many folks had with GetVersion was the need to break these values out of the single returned Long integer, as major and minor version numbers of both Windows and DOS were stored in individual bytes. Though it is more tidy having these numbers stored within their own elements of the OSVERSIONINFO structure, one oddity remains. The build number is stored within the low word of the dwBuildNumber element. No documentation exists for what's returned in the high word.

Two other properties, Version and Platform, return string representations and might prove useful in an About message box or something similar. I wrote the Platform property to return the name of the OS, either "Windows 95" or "Windows NT." The Version property appends all available data to the name, such as "Windows 95 v4.0, Build 950." One other element of OSVERSIONINFO, szCSDVersion, is undocumented and doesn't currently provide any useful information.

PASSING CONTROL ARRAYS IN VB3

One of the more frequent questions I've seen online is, "How do I pass a control array to a subroutine or function?" Strangely enough, you can't. At least, not directly. In order to pass a control array, you must first convert it to an array of controls. Confused yet? Though it sounds (and is!) pretty twisted, the implementation remains fairly easy.

I wrote a little demo program that has a control array consisting of 10 label controls on the main form. Each label control has a different BackColor assigned to it and no Caption. This group of label controls functions as a makeshift graph of sorts. By changing the width of the labels, you can make them look like a horizontal bar chart (see Figure 3). For the purpose of demonstrating, when the user clicks on the form, the Form_Click event converts the control array to an array of label controls, which is then passed to another routine that actually changes the displayed data:

```
Sub Form_Click ()
   ReDim Graph(0 To 9) As Label
   Dim i As Integer

   For i = LBound(Graph) To UBound(Graph)
      Set Graph(i) = lblGraph(i)
   Next i
   DoGraphDisplay Graph()
End Sub
```

The called routine, DoGraphDisplay, can then iterate through the array of controls, adjusting each label's width appropriately (or, for this demo, randomly). To make the demo interesting, I've written the DoGraphDisplay routine to adjust each label 30 times, giving the appearance of some sort of real-time monitoring chart:

```
Sub DoGraphDisplay (Bars() As Label)
   Dim min As Integer, max As Integer
   Dim i As Integer, j As Integer

   min = Screen.TwipsPerPixelX
   max = Bars(0).Parent.ScaleWidth - Bars(0).Left * 2
   For i = 0 To 29
      For j = LBound(Bars) To UBound(Bars)
         Bars(j).Width = Rnd * (max - min) + min
         Bars(j).Refresh
      Next j
   Next i
End Sub
```

This same method also works just fine in VB4, although the newer version of VB adds some interesting alternative methods as well. You may wish to explore passing the control array directly, but you will need to declare the receiving routine's parameter As Variant. ×

Code Online

For all the current issue's listings in one file, go to the Registered Level of The Development Exchange (http://www.windx.com), The Microsoft Network (GO WINDX), or CompuServe* VBPJ *Forum's Magazine Library (GO VBPJ). Development Exchange Premier Level subscribers ($20 for six months) can get each article's listings in a separate file, as well as additional code and utilities for selected articles, plus archives of all code ever published in* VBPJ *and* Microsoft Interactive Developer *magazines.

Unicode Strings You Along

Locator+ Codes

Listings ZIP file (free Registered Level): VBPJ0996
Listings for this article (subscriber Premier Level): PT0996P