Visual Basic 4.0 Technical Articles

Creating Useful Native Visual Basic and Microsoft Access Functions

Kenneth Lassesen
Microsoft Developer Network Technology Group

Created: August 26, 1994

Abstract

This article describes how you can add compiled C functions to Microsoft® Visual Basic® and Microsoft Access®—functions that appear to be native functions in Visual Basic rather than application programming interface (API) functions being called from Visual Basic. This approach results in fewer lines of code, less coding time, and better performance. The examples in this article and in the accompanying sample files show how structures, pointers to pointers, and C++-related agony can be avoided by the Visual Basic and Microsoft Access programmer.

What Are Native Functions?

Programmers of Microsoft® Access® and Visual Basic® are familiar with two styles of functions: native functions and application programming interface (API) functions. Native functions will allocate memory automatically without the programmer's intervention. For example, here are typical lines of Basic code:

```
ThisPath$=Environ ("Path") 'The string returned is created by the function and
A$=dir("*.txt")           'the programmer does not need to allocate memory first.
```

API functions can be either entry points into the operating system architecture or special functions not visible in the application language. API functions do not allocate memory space, but require the programmer to create the space for them. API functions called by preallocating space use what I call a C-like method. For example:

```
cbBufferLen% = 255              'Determine maximum length of string returned.
SzbUffer = Space(cbBufferLen%)
rc% = GetProfileString("intl", "sLongDate", "-1", SzbUffer, cbBufferLen%)'Make API Call
If rc% > 0 Then                 'rc% contains the NUMBER OF CHARACTERS in string
    WinIniValue$ = Left(SzbUffer, rc%) 'Truncate garbage off end of string
Else
    WinIniValue$ = ""           'Failed or no string
End If
```

This style of coding is exceedingly long, can be confusing, and requires lengthy debugging. For example, the above function call, GetProfileString, will fail with some versions of the declaration statement for it.

This C-like method is contrary to the way Basic programmers code. Each line of code should do one useful unit of work at a higher level of abstraction than in C. For example, each line below in this filling of a list box would be a routine or several lines of code in C:

```
Open "Mydata.dat" for input as #fno    'Open file.
While not eof(fno)                     'Is this the end?
   Line Input #fno, A$                 'No, read next line.
   ListBox1.AddItem A$                 'Place in listbox.
WEND                                   'Repeat until end of file.
Close #fno                             'Close file.
```

A function to obtain a value from an .INI file should ideally be called as:

```
IniValue$=GetWinIni("Fonts","Arial")  'Get location of Arial font file.
```

This way of obtaining a value means no memory allocation or cleanup by the Basic programmer and constitutes one line of code for one useful unit of work.

Functions that exhibit this latter style I say have the Visual Basic–native method of calling because, although the function is contained in a dynamic-link library (DLL), it returns a Visual Basic string, and the way it is called from Visual Basic appears to be the same way other Visual Basic functions are called. In other words, it is natural or native.

History

When I started programming in Visual Basic version 1.0, I developed a library of functions coded in Visual Basic and placed their code in modules that were used time and time again—my Visual Basic library. Work pressures prevented me from converting this library into a DLL written in the C language as I slaved away generating Visual Basic and Microsoft Access applications. When I sat down to rewrite the code in C, I found that C could easily handle the Microsoft Windows® structures and return numeric values from them. Normally, C cannot return a string from a function unless it is given space by the program calling it. The returning of a string from a DLL appeared to be impossible using a Visual Basic–native approach. The choices appeared to be as follows:

- Allocate space in Visual Basic first and then call the function, but this is ugly and a kluge.
- Change the functions into properties of a Visual Basic control (VBX), so space can be allocated from the VBX memory segment.
- Pass multiple arguments using a VBX and, therefore, set properties one at a time—another kluge.

Given these unappealing options, I was determined to find a coding solution that is natural to Visual Basic. The answer was the VBCreateTempHlstr function in the Visual Basic version 3.0 Control Development Guide (Product Documentation, Office Developer's Kit 1.0, Visual Basic 3.0 Professional Edition). This function creates space in the Visual Basic memory space which Visual Basic recovers automatically when the handle is passed back to Visual Basic. This function thereby allows the development of Visual Basic–native functions, which allows me to exchange the Visual Basic-module-based library of functions for a C-version DLL. As my Microsoft Access work increased, I speculated that Microsoft Access would borrow significantly from Visual Basic and was delighted to discover by experiment that Visual Basic–native functions appear to work with both languages. Figure 1 shows the difference between these methods.

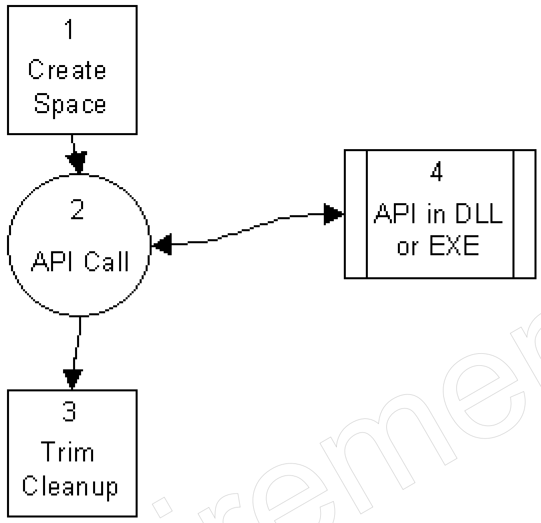

Figure 1. C-like method

In the C-like method, memory and structures must be created and initialized before the API call and cleaned up afterwards, all in Visual Basic code (Figure 2).

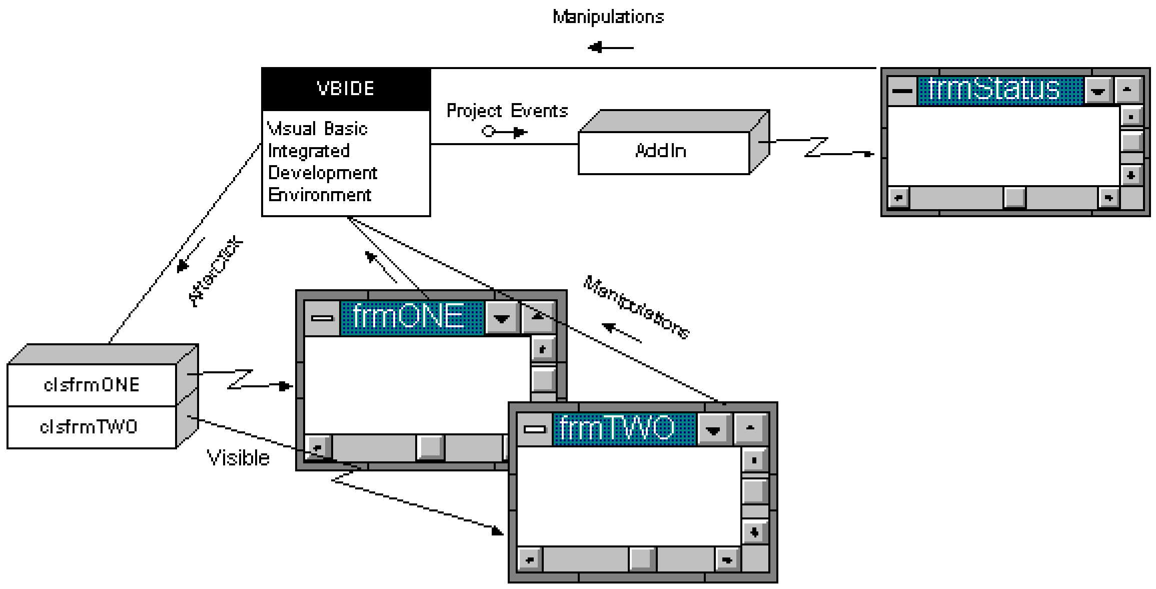

Figure 2. Visual Basic–native method

In the Visual Basic–native method, Visual Basic does not need to create or initialize memory, nor clean up afterwards. The intermediate DLL handles any structures or memory requirements and obtains space from Visual Basic using the VBCreateTempHlstr function. The Visual Basic engine (VBRUNx00.DLL) automatically handles the cleanup of this space.

Although the Visual Basic–native method appears to be more complicated, it is very simple when used from Visual Basic. Any API call is reduced to a single declaration and a one-line call, no matter how complex the API is. The rest of the work is in the intermediate DLL away from the Visual Basic programmer.

With this approach, it's apparent that a rich collection of native functions can easily be added to Visual Basic and Microsoft Access. This article shows examples of various styles of functions that can appear as Visual Basic–native functions rather than C-like functions.

> Note   The use of VBCreateTempHlstr with Microsoft Access version 1.1 or 2.0 is not officially supported nor documented and may not work for future versions of Microsoft Access.

Splitting the Clock Tick

Code performance has always been a concern of professional programmers. In my MS-DOS® days, I ended up programming clock chips to accurately measure the performance of code. In Windows this is not so simple and risks ugly side effects from using timers already in use by other applications. Many Visual Basic programmers believe that under Windows the accuracy of timing is 55 milliseconds (1 clock tick)—the accuracy of time returned from Now or Timer. You can get true millisecond accuracy very easily by using the correct function—but you may have the overhead of calling it in Visual Basic.

Our first example of a native Visual Basic function shows how structures can be hidden from the Visual Basic programmer. I encapsulated this function call to reduce the number of lines of Visual Basic code and improve readability. The following two routines encapsulate TimerCount, one of the Windows API timer functions that requires a structure.

- `Sub StopWatch_Reset` resets the stopwatch to 0 milliseconds. This occurs automatically on the first call.
- `Function StopWatch_Time() as Long` reports the number of milliseconds since either the this function was first called or StopWatch_Reset was last called.

The TimerCount function is in TOOLHELP.DLL, which can be used with Windows version 3.0. TimerCount's functionality is also provided by timeGetTime in MMSYSTEM.DLL (available from Windows 3.1 and Windows NT™), which does not require a structure and would be preferred in practice.

C Coding

When a Visual Basic application needs to obtain information from Windows, a structure is often required for the API call. The steps to implement this call are as follows:

1. Convert the C structural declaration into the equivalent Visual Basic structure.
2. Initialize the structure if required.
3. Call the function.
4. Check the results.
5. Extract the data.

This process may add 10, 20, or 50 lines of code to an application. Frequently, this code is needed again in the next project, causing the programmer to rewrite the code and test it again, or locate and copy it from another project. Our first example of a Visual Basic–native method function places the code in a DLL that returns only the information requested and requires only one line of code to use.

The C code for the two functions described above—StopWatch_Reset and StopWatch_Time—is shown below. The StopWatch_Reset function initializes the structure and keeps track of the exact time it was called so that the milliseconds elapsed since it was called can be reported.

```
#include <toolhelp.h>   //For StopWatch
TIMERINFO tiThisTime;
DWORD dwStopWatchStarted=0;
//------------------------------------------------------------------
//  .StopWatch_Reset()
//------------------------------------------------------------------
// We set StopWatchStarted to the current ticks.
void  _pascal __export StopWatch_Reset()
{
if (tiThisTime.dwSize==0)
   tiThisTime.dwSize=sizeof(TIMERINFO);
if (0==TimerCount(&tiThisTime))
   MessageBox(NULL,"Unable to obtain current time for StopWatch.",
              "Unexpected Error",MB_ICONSTOP);
   dwStopWatchStarted=tiThisTime.dwmsThisVM; //Records starting millisecond.
}
```

The second function, StopWatch_Time, calls TimerCount and returns the number of milliseconds since StopWatch_Reset was called.

```
//-----------------------------------------------------------------
//  .StopWatch_Time()
//-----------------------------------------------------------------
// We return the ticks since StopWatchStarted was reset above.
long _pascal __export StopWatch_Time()
{
if (dwStopWatchStarted==0)  //If user forgot to set it, we reset it.
      StopWatch_Reset();
if (0==TimerCount(&tiThisTime)) //Get time
   MessageBox(NULL,"Unable to obtain current time for StopWatch.",
              "Unexpected Error",MB_ICONSTOP);
return  (long) (tiThisTime.dwmsThisVM-dwStopWatchStarted);
// Returns milliseconds since start.
}
```

The code cooperates with Visual Basic programming. Visual Basic has no simple mechanism for detecting an error that occurs in a DLL. Error information from the DLL can be returned by a parameter that is set to inform the program or by a message box that informs the user.

If the parameter approach is used, most programmers would need to check for the error condition and then display a message box containing their own message. I favor the message box option over the parameter one for two reasons:

- It requires less code in Visual Basic.
- It is more informative and gives complete error reports to users.

Many Visual Basic programmers fail to do all of the appropriate checks on return values and usually add error handling after the error has occurred the first time. Visual Basic programmers should never be counted on to call things properly or in order. If some step should be done and the programmer has forgotten to do it, the function should do it for them automatically rather than give an error message, as in the calling of StopWatch_Reset above.

The code below shows both methods of calling TimerCount and its alternative function, timeGetTime.

```
Declare Function TimerCount Lib "ToolHelp.DLL" (tagTimerInfo As Any) As Integer
Declare Function StopWatch_Time Lib "NATIVEVB.DLL" () As Long
Type tagTimerInfo
    dwSize As Long
    dwStart As Long
    dwVM  As Long
End Type

Dim i As Integer, b    As Integer, j As Integer
Dim TimerInfo As tagTimerInfo
TimerInfo.dwSize = Len(TimerInfo)

'Find out how many milliseconds to execute 1000 times using the C-like method.
t1& = Stopwatch_time()
For j = 1 To 1000
    rc% = TimerCount(TimerInfo)
    T& = TimerInfo.dwVM
Next j
t2& = Stopwatch_time()         'Difference between this and t1& is milliseconds
                               'to execute loop.
'Find out how many milliseconds to execute 1000 times using the
'VB-native method.
For j = 1 To 1000
    T& = Stopwatch_time()
Next j
t3& = Stopwatch_time()         'Difference between this and t2& is millseconds
                               'to execute.
For j = 1 To 1000

Next j
t4& = Stopwatch_time()
For j = 1 To 1000
    T& = timegettime()
Next j
t5& = Stopwatch_time()
MsgBox "C-Like Method:" & Str$(t2& - t1&) & "  VB-Native Method:" & Str$(t3& -
t2&) & "  Loop time:" & Str$(t4& - t3&) & " timeGetTime:" & Str$(t5& - t4&) & ""
```

This code also allows comparisons among the C-like, Visual Basic–native, and direct-API-call methods. We repeatedly ran the code above, producing the following times:

- calls for the C-like method: 249 milliseconds with standard deviation of 9.5
- calls by Visual Basic–native method: 208 milliseconds with standard deviation of 6.7
- calls by timeGetTime method: 131 milliseconds with standard deviation of 5.7

The performance of the Visual Basic–native method approach was marginally better than the C-like method, despite the additional functionality and checking for errors. Using the timeGetTime method was fastest, but did not include any checking for errors. You may wish to try this with your PC; the Visual Basic project Perf0.Mak in the NATIVEVB sample that accompanies this article contains the source code.

> Note For accurate timing, remember to unload other applications from Windows and disconnect from the network to reduce sources of variance.

Example of Using Stopwatch

The following code sample shows how we can use these new native functions to monitor performance in our Visual Basic and Microsoft Access code. It is important that we remove as much measurement overhead as possible, so we'll use an array to store our data while timing and then write the data out for later analysis. If you are on a network, it is recommended that you restart Windows without the network (that is, type win /n or disconnect your network card before running the included LOAD_CTL.MAK project). A sample of one procedure is shown below:

```
Declare Function STopWatch_Time Lib "NATIVEVB.DLL" () As Long
Dim Datapts(0 To 1000)          As Long

Dim i As Integer, j As Integer, fno As Integer
On Error GoTo Combo1_Unload
Datapts(0) = StopWatch_Time()
For i = 1 To 1000
    Load Combo1(i)'Load 1000 copies or until memory out.
    CtlMax% = i
    Datapts(i) = StopWatch_Time()
Next i

Combo1_exit:
Exit Sub
Combo1_Unload:
On Error Resume Next
For j = CtlMax% To 1 Step -1
    Unload Combo1(j)'Load 1000 copies or until memory out.
Next j
LogData "Combo1", i%
Resume Combo1_exit
```

The LOAD_CTL.MAK project writes the information to a tab-delimited text file that can be imported into Microsoft Excel or Microsoft Access. The chart below shows the very significant difference between load time and number of instances allowed of Visual Basic image controls and Visual Basic picture controls. I suspect everybody will be profiling the code and load times for their favorite controls after reading this.

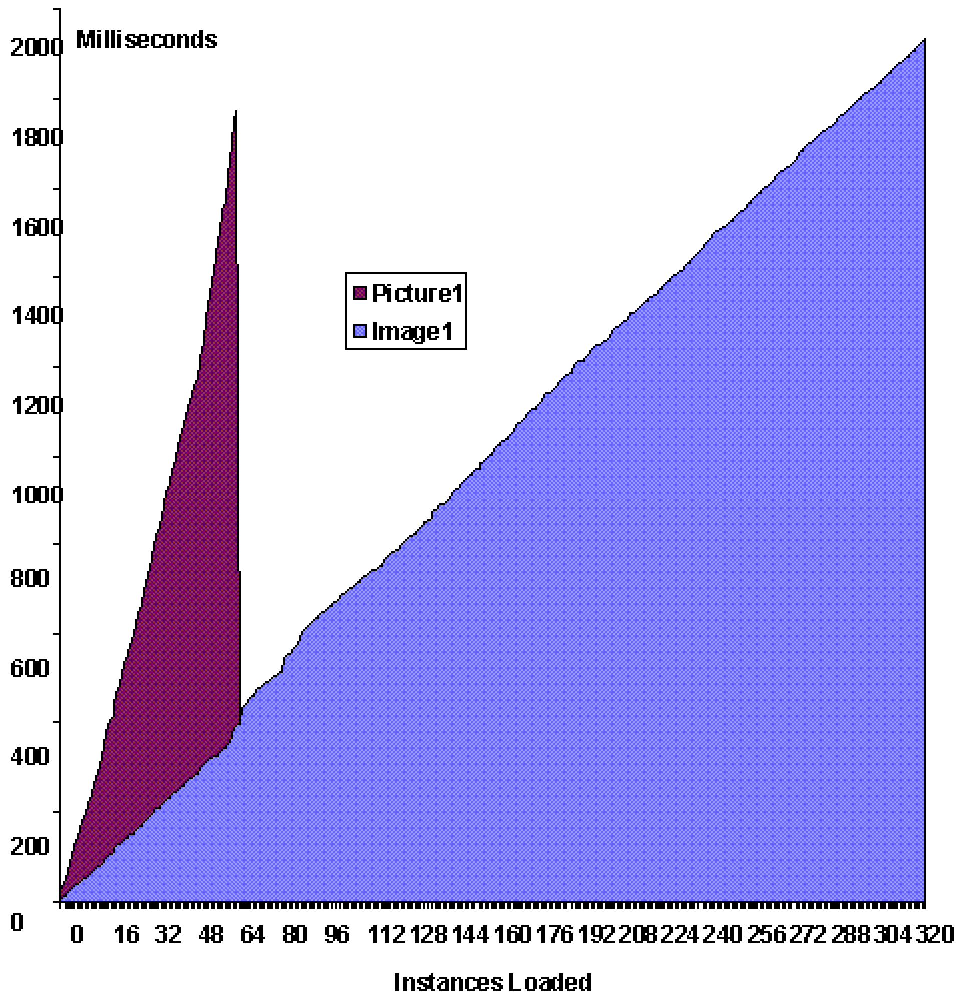

Figure 3. Load times for image and picture controls

What Is the Advantage?

The Visual Basic–native method is simply a C wrapper around the call of a DLL, so why go to the trouble of writing a DLL? Some benefits of writing a DLL are the following:

- Better performance—that is, less time per call, as well as more accurate results.
- Less code to write. The Visual Basic–native method of calling the API requires a single statement for a declaration and a single statement per use. The C-like method requires 8 declaration/initialization statements and 2 statements per use.
- The Visual Basic–native method reduces the need for a Visual Basic programmer to know the operating system function calls.

Initialization Functions

Initialization functions such as GetProfileString are not the simplest functions to use successfully from Visual Basic's perspective, despite their apparent simplicity. A brief review of the Knowledge Base reveals many articles on using the initialization functions and Visual Basic: Q105807, Q75639, Q110219, Q109290, Q86470, and Q69888. Several shareware products are available that solely handle .INI files—a reflection of the frustration many Visual Basic users have had using the GetProfileString function.

The code below is a robust implementation of the GetPrivateProfileString call. It verifies that appropriate arguments are passed in—a needed caution, given the use of variants in Visual Basic and multiple ways of declaring calls from Visual Basic. The function then makes the call using its own buffer (iniBuffer) and checks for error conditions. Then the function creates a Visual Basic string to return to Visual Basic via VBCreateTempHlstr.

```
#define MAXINISTRING 4096 //Largest string that can be returned
#define EmptyString "\0"
char iniBuffer[MAXINISTRING];
HLSTR hlstr;
HLSTR __export  _pascal vbGetIni(LPSTR Section, LPSTR Entry, LPSTR FileName)
{int cb;
      if((Section==NULL) || (Entry==NULL))
        {
         MessageBox(NULL,"Section or Entry is NULL -- wrong function
                    called.","GetWinIni()",MB_ICONSTOP);
         return NULL;
         }
      cb = GetPrivateProfileString(Section, Entry, EmptyString, iniBuffer,
                                   MAXINISTRING, FileName);
      if (cb < 1)
         return NULL ;
      if (cb ==MAXINISTRING)
            MessageBox(NULL,"Value may be
                       truncated.","vbGetIni()",MB_ICONEXCLAMATION);
      hlstr = VBCreateTempHlstr(iniBuffer, cb);
     if (HIWORD(hlstr) == -1)
     { MessageBox(NULL,"VBCreateTempHlstr failed -- Contact Support." ,

                  "CRITICAL ERROR",MB_ICONSTOP);
         return NULL;}
     return hlstr;
}
```

The code follows normal conventions of programming for Windows until the last few lines, where memory is allocated from Visual Basic, lines that you will see again and again:

```
     hlstr = VBCreateTempHlstr(iniBuffer, cb);
     if (HIWORD(hlstr) == -1)
     { MessageBox(NULL,"VBCreateTempHlstr failed -- Contact Support." ,
                  "CRITICAL ERROR",MB_ICONSTOP);
         return NULL;}
```

This obtains memory space from Visual Basic and copies the needed characters into it. The purpose of the test after copying is to validate that space has been successfully allocated. These lines can be placed in a macro to reduce coding. For example:

```
#define VALIDHLSTR(x)  if (HIWORD(x) == -1)
 { MessageBox(NULL,"VBCreateTempHlstr failed -- Contact Support." ,
              "CRITICAL ERROR",MB_ICONSTOP);return NULL;}
```

VBCreateTempHlstr and HLSTR

The Visual Basic version 3.0 Control Development Guide (Product Documentation, Office Developer's Kit 1.0, Visual Basic 3.0 Professional Edition) has a function contained in VBAPI.LIB called VBCreateTempHlstr, which permits a DLL to create a Visual Basic–style temporary string. The string space is obtained from Visual Basic, which allows a maximum of 20 temporary strings. These strings are automatically deleted the first time they are used by Visual Basic. Unlike many other functions in the CDK, this function does not require you to explicitly destroy the memory allocated; Visual Basic destroys it for you after Visual Basic copies the contents into its own memory space.

People who have not developed VBXes are unfamiliar with what an HLSTR is. It is a handle to a Basic language string. These are roughly similar to Pascal strings—the size is stored separately from the characters—without a terminating Chr$(0). This format permits any character, including Chr$(0), to be in a string. The code handling of a HLSTR is different from an LPSTR that terminates with Chr$(0). Calling functions that pass the wrong type from Visual Basic can create nasty problems; for example, if you send an HLSTR instead of an LPSTR because of an incorrect declaration in Visual Basic, you may find that the string could be several megabytes long instead of 4 bytes because the first Chr$(0) may be in another application's memory space—nasty!

Naming standards

In the examples in the rest of this article, you will note that some functions use vb before the name. This signifies that this function only works with Visual Basic and Microsoft Access. Care must be taken in the Visual Basic declarations because HLSTR and LPSTR are very different creatures.

Performance results

Now that we have the first function using HLSTR, let us test it against the C-like method, the Visual Basic–native method, and a shareware VBX that provides the same functionality. The code below was used to create Table 1. When measuring performance, it is important to begin on the second or subsequent pass of code because components like Windows or SMARTDRV caching may affect the results. You may wish to try this with your PC. The Visual Basic PERF1.MAK project in the NATIVEVB sample contains the source code. The unsupported VBNATIVE tool that accompanies the related article "A Collection of Useful Native Visual Basic and Microsoft Access Functions" (MSDN Library Archive, Technical Articles) has all the initialization functions, including code.

```
Declare Function vbGetIni Lib "NATIVEVB.DLL" (ByVal Section$, ByVal KeyWord$,
                                             ByVal Filename$) As String

Dim i As Integer, b As Integer, j As Integer
Section$ = "TEST"
FileIni$ = "C:\TEST.INI"
For i = 1 To 100    'Fill a dummy .INI file with 100 values.
    Ent$(i) = Format(i, "0")
    PutIni Section$, Ent$(i), Ent$(i), FileIni$
Next i
Init1.Application = "Test"  'Get Shareware VBX an advantage.
For b = 0 To 1    'We do a dummy pass to compensate for caching.
 For i = 0 To 2    'Method to try
  For j = 1 To 100    'Retrieve all values.
    Select Case i    'Select method.
    Case 0    'C-like method
        Buffer$ = String(128, " ")
        rc% = GetPrivateProfileString(Section$, Ent$(j), "",
                                      Buffer$, 4096, FileIni$)
        A$ = Left$(Buffer$, rc%)
    Case 1    'VB-native method
        A$ = vbGetIni(Section$, Ent$(j), FileIni$)
    Case 2    'Shareware VBX
       Init1.Parameter = Ent$(j)
       A$ = Init1.Value
    End Select
    If b = 1 Then TimeData(i, j) = STOPWATCH_Time()
  Next j
 Next i
Next b
T$ = Chr$(9)    'Tab character so easy to read into Excel
fno = FreeFile
Open "C:\TEST.TXT" For Output As #fno
For j = 1 To 100    'Write the elapsed time only.
    Print #fno, TimeData(0, j) - TimeData(0, j - 1); T$,
       TimeData(1, j) - TimeData(1, j - 1); T$, TimeData(2, j)
        - TimeData(2, j - 1)
Next j
Close
MsgBox "Timing complete"
End
```

The bottom line of Table 1 displays the average number of milliseconds it takes to read 100 values from a .INI file. The data in the table is a sample from these 100 timings. As you can see from the bottom line of the table, the Visual Basic–native method provided almost a 500 percent improvement in performance over the C-like method. The shareware VBX was twice as slow as the C-like method.

Table 1. Performance Comparison in Milliseconds of Three Methods

C-like method	Visual Basic–native method	VBX
5	1	10
5	0	11
9	0	9
6	1	12
7	2	11
4	1	10
6	1	12
8	0	10
5	0	8
6	2	11
4	2	10
5	0	9
6	1	7
4	1	7
9	0	90
4	1	10
6	0	51
6.08 Avg.	1.22 Avg.	11.29 Avg.

Version Information

Version information is often critical for diagnosing problems when an application is distributed in a corporation. If you read the Setup Wizard code, you'll see a classic use of the C-like method to get this information; using the Visual Basic–native method approach is much simpler. I have frequently found that other applications load different versions of DLLs and VBXes. If these DLLs and VBXes are already in memory when my application is launched, they may affect my programs, so I would like to have version information easily available when my application is running.

I hate reinventing the wheel, so I looked in the MSDN Library Archive for an example using version information. VERSTAMP (MSDN Library Archive, Sample Code; search for VERSTAMP), contains a project demonstrating how to get version information. In the following section, I will take you through the process of converting VERSTAMP's sample code into a Visual Basic–native method function.

The Fast Sample Hack

VERORIG.C contains the following procedures: WinMain, WndProc, About, MyGetOpenFileName, ShowVerInfo, ClearDlgVer, FillVerDialog, MyVerFindFile, MyVerInstallFile, MoreVerInfo, HandleVerFindFileRes, PostInstallProcessing, and HandleVerInstallFileRes. After reading their contents, I deleted all of these procedures except MoreVerInfo, ShowVerInfo, and FillVerDialog. These routines access the functions I need, but place the values I want in a dialog box instead of returning these values. For example, this code places a string in a dialog box:

```
lstrcpy(gszUserMsg, "Unknown");
....
SetDlgItemText(hWnd, ++wDlgItem, gszUserMsg);
```

Instead of filling a dialog box with strings, we want to return the values to the calling program as HLSTRs. This is done by replacing the last line above with the three lines shown below:

```
hlstr = VBCreateTempHlstr(gszUserMsg, strlen(gszUserMsg));
VALIDHLSTR(hlstr);
return hlstr;
```

The function calls were changed as shown below.

Before	`BOOL ShowVerInfo(HWND hWnd, LPSTR szDir, LPSTR szFile, WORD wDlgItem)`
After	`HLSTR ShowVerInfo(LPSTR szFullPath, int item)`
Before	`void FillVerDialog(HWND hWnd, VS_VERSION FAR *pVerInfo, WORD wDlgItem)`
After	`HLSTR FillVerDialog( VS_VERSION FAR *pVerInfo, int I)`
Before	`BOOL FAR PASCAL __export MoreVerInfo (HWND hDlg, unsigned wMsg, WORD wParam, LONG lParam)`
After	`HLSTR FAR PASCAL MoreVerInfo (LPSTR szFullPath, int I)`

Because we want the function to return information rather than fill a dialog window, we must change the functions so that they return LHSTR and drop the HWND parameters in the call. The LPSTR szDir and LPSTR szFile parameters were replaced with LPSTR szFullPath, similar to the value returned by the common dialog control (COMMDLG.VBX) filename property. The __export reserved word was removed from a call I did not want visible.

```
#include <vbapi.h>         // For HLSTR and so forth
#include  <string.h>       // For strlen
#include "NativeVB.h"      // For VALIDHLSTR and so forth
```

The example filled multiple dialog boxes with information, but I wanted it to return a single HLSTR. The simple solution was to add a switch statement so that I could specify which piece of information I wanted returned. The code was originally:

```
// Fill in the file version.
  wsprintf(gszUserMsg,
           "%d.%d.%d.%d",
           HIWORD(pVerInfo->vffInfo.dwFileVersionMS),
           LOWORD(pVerInfo->vffInfo.dwFileVersionMS),
           HIWORD(pVerInfo->vffInfo.dwFileVersionLS),
           LOWORD(pVerInfo->vffInfo.dwFileVersionLS));
  SetDlgItemText(hWnd, wDlgItem, gszUserMsg);

  // Fill in the product version.
  wsprintf(gszUserMsg,
           "%d.%d.%d.%d",
           HIWORD(pVerInfo->vffInfo.dwProductVersionMS),
           LOWORD(pVerInfo->vffInfo.dwProductVersionMS),
           HIWORD(pVerInfo->vffInfo.dwProductVersionLS),
           LOWORD(pVerInfo->vffInfo.dwProductVersionLS));
  SetDlgItemText(hWnd, ++wDlgItem, gszUserMsg);

  // File flags are bitwise or'ed so there can be more than one.
  // dwNum is used to make this easier to read.
  dwNum = pVerInfo->vffInfo.dwFileFlags;
  wsprintf(gszUserMsg, "%s %s %s %s %s %s %s",
          (LPSTR) (VS_FF_DEBUG         & dwNum ? "Debug"   : ""),
          (LPSTR) (VS_FF_PRERELEASE    & dwNum ? "PreRel"  : ""),
          (LPSTR) (VS_FF_PATCHED       & dwNum ? "Patched" : ""),
          (LPSTR) (VS_FF_PRIVATEBUILD  & dwNum ? "Private" : ""),
          (LPSTR) (VS_FF_INFOINFERRED  & dwNum ? "Info"    : ""),
          (LPSTR) (VS_FF_DEBUG         & dwNum ? "Special" : ""),
          (LPSTR) (0xFFFFFF00L         & dwNum ? "Unknown" : ""));
  SetDlgItemText(hWnd, ++wDlgItem, gszUserMsg);
```

With the added switch statement, the code became:

```
switch (i)
 {
   case 9:
     // Fill in the file version.
     wsprintf(gszUserMsg,
           "%d.%d.%d.%d",
           HIWORD(pVerInfo->vffInfo.dwFileVersionMS),
           LOWORD(pVerInfo->vffInfo.dwFileVersionMS),
           HIWORD(pVerInfo->vffInfo.dwFileVersionLS),
           LOWORD(pVerInfo->vffInfo.dwFileVersionLS));
           hlstr = VBCreateTempHlstr(gszUserMsg, strlen(gszUserMsg));
           VALIDHLSTR(hlstr);
          return hlstr;

   case 10:
     // Fill in the product version.
     wsprintf(gszUserMsg,
           "%d.%d.%d.%d",
           HIWORD(pVerInfo->vffInfo.dwProductVersionMS),
           LOWORD(pVerInfo->vffInfo.dwProductVersionMS),
           HIWORD(pVerInfo->vffInfo.dwProductVersionLS),
           LOWORD(pVerInfo->vffInfo.dwProductVersionLS));
           hlstr = VBCreateTempHlstr(gszUserMsg, strlen(gszUserMsg));
           VALIDHLSTR(hlstr);
          return hlstr;
   case 11:
     // File flags are bitwise or'ed so there can be more than one.
     // dwNum is used to make this easier to read.
     dwNum = pVerInfo->vffInfo.dwFileFlags;
     wsprintf(gszUserMsg, "%s %s %s %s %s %s %s",
          (LPSTR) (VS_FF_DEBUG         & dwNum ? "Debug"   : ""),
          (LPSTR) (VS_FF_PRERELEASE    & dwNum ? "PreRel"  : ""),
          (LPSTR) (VS_FF_PATCHED       & dwNum ? "Patched" : ""),
          (LPSTR) (VS_FF_PRIVATEBUILD  & dwNum ? "Private" : ""),
          (LPSTR) (VS_FF_INFOINFERRED  & dwNum ? "Info"    : ""),
          (LPSTR) (VS_FF_DEBUG         & dwNum ? "Special" : ""),
          (LPSTR) (0xFFFFFF00L         & dwNum ? "Unknown" : ""));
           hlstr = VBCreateTempHlstr(gszUserMsg, strlen(gszUserMsg));
          VALIDHLSTR(hlstr);
          return hlstr;
```

The switch operates on a passed-in parameter that specifies the information to be returned. This allows the code to stay in the same structure as originally written—"If it ain't broke, don't recode it."

Putting It All Together

We have most of the version information function completed, with the exception of adding any features to make it easier to use from Visual Basic. The VERSTAMP example, VERORIG.C, has two routines that return data—and inasmuch as we want to keep the code structured like the example, we'll add an entry-point function to point us to the correct routine.

```
HLSTR _pascal __export vbGetVerInfo(LPSTR szFullPath, int item)
{
   if (item <=8)

      //0 "Illegal string"     ,
      //1 "CompanyName" //These are all handled by the same call.
      //2 "FileDescription"
      //3 "FileVersion"
      //4 "InternalName"
      //5 "LegalCopyright"
      //6 "LegalTrademarks"
      //7 "ProductName"
      //8 "ProductVersion
      return ShowVerInfo(szFullPath,item);
   else
         return BasicVerInfo(szFullPath,item);
   }
}
```

Some files may not have version information. An empty string returned above does not imply that there is no version information string; that element may simply have been left blank. To determine if version information is available, I created a second function, HasVerInfo.

In Visual Basic the code might appear as follows:

```
CMDialog1.Filename = "*.*" 'Look for all files.
CMDialog1.DialogTitle = "Select file to obtain version info from."
CMDialog1.Flags = &H1800&  'Selected file must exist.
CMDialog1.Action = 1      'OpenFile action
vsinfo(i) = CMDialog1.Filename
If HasVerInfo(CMDialog1.Filename) Then 'Should we call vbGetVerInfo?
    For i = 1 To 15                    '15 pieces of VerInfo are available.
        vsinfo(i) = vbGetVerInfo(CMDialog1.Filename, i)
    Next I
Else
   For i = 1 To 15 'There are 15 pieces of VerInfo available.
        vsinfo(i) = "n/a" 'Inform user that there is no info.
    Next I
End if
```

To implement the function HasVerInfo in our DLL, we add:

```
BOOL __export _pascal HasVerInfo(LPSTR szFullPath)
{
  DWORD dwVerInfoSize;
  DWORD dwVerHnd;

                   // You must find the size first before getting any file info.
  dwVerInfoSize =GetFileVersionInfoSize(szFullPath, &dwVerHnd);
  if (dwVerInfoSize)  'If size is 0, no information is there.
    {
    return TRUE;   // Return success
    }
  else
    {
    return FALSE;                    // Return failure
    }
}
```

The source code in NATIVEVB.DLL, which accompanies this article, applies the same techniques to other encapsulated functions. For homework, locate the original code of VERSTAMP in the MSDN Library Archive and compare to see how the conversion was done. The OLE Summary Information functions in the sample code demonstrate how very complex processing can be handled quite simply from Visual Basic's perspective.

VBX or Visual Basic–Native Method?

The version information function represents a style of functions that uses structures. These structure-based functions require allocation of memory and navigation to obtain the information—why not encapsulate these functions in a VBX that has all the version information properties? Unless there are significant demonstrated performance differences, I would discourage this encapsulation approach for the following reasons:

- The VBX will definitely be slower if you need only a few properties.
- You have to add the VBX to the project and place an instance on a form.
- You no longer have dual Visual Basic and Microsoft Access functions.
- Encapsulation requires more lines of C code.

As a general principle, never write a VBX or OLE Control (OCX) that does not have events. DLLs can provide the same functionality as a VBX or OCX with better performance in most cases.

Going in the Other Direction

We have discussed functions that take C-like parameters and return an HLSTR. Visual Basic strings can contain Chr$(0) or \0. When we pass a string with Byval A$, Visual Basic creates a copy of the string and then passes its pointer to the function. This string is a C string or LPSTR; that is, it is terminated by the first Chr$(0). If the Visual Basic string happens to have a Chr$(0) in it, any characters after Chr$(0) will not be processed by the function. This problem can be resolved by passing in the Visual Basic string as an HLSTR instead of as an LPSTR. An LPSTR is a C string that terminates with a \0 or Chr$(0).

For example, if you open a file in binary mode, you can place the data into a Visual Basic string simply by reading it. Any manipulation of this string of binary data must be done using Visual Basic functions that accept the presence of Chr$(0) in the string. This manipulation can be slow, making a C-coded function preferable.

Count the Nulls

Some API calls like GetProfileString return a string with a Chr$(0) between elements and Chr$(0)+Chr$(0) at the end of an array. It would be nice to be able to count the NULLs in the string. The trick is to pass the Visual Basic string in as a HLSTR not as a LPSTR. This means that the call is not like this:

```
Visual Basic:    Declare Function Lib "NATIVEVB.DLL" vbCountNulls(Byval Buffer$) as Integer

C:               int __export _pascal vbCountNulls(LPSTR buffer)
```

Instead, the call should be like this:

```
Visual Basic:    Declare Function Lib "NATIVEVB.DLL" vbCountNulls(Buffer$) as Integer

C:               int __export _pascal vbCountNulls(HLSTR buffer)
```

The latter call allows the string to be passed in as an HLSTR whereas the former will have the string passed in as an LPSTR. The string may not be the complete string because the first Chr$(0) will truncate it.

Care must be taken with the string passed into a DLL as an HLSTR because any changes made to it in the DLL may be reflected in the original string in Visual Basic.

The code to implement vbCountNulls is shown below. Instead of waiting for *lpstr=='\0' to identify when the end of the string is reached, a counter is used. The key to working with an HLSTR is obtaining the length and then knowing when you have reached the last character.

```
int __export _pascal vbCountNulls(HLSTR hlstr)
{USHORT i=0, cb=VBGetHlstrLen(hlstr);  //Obtain length of string.
 int cnt=0;   //Set count to zero.
 LPSTR lpStr=VBDerefHlstr(hlstr);
 while(i++ < cb)  //Are we at the end of the string?
    if(*lpStr++=='\0') cnt++;  //If a Chr$(0), increment counter.
 return cnt;
}
```

AllTrim

In Visual Basic, the need for trimming spaces off both ends of a string is often solved by Trim. This function does not handle other white space, such as embedded tabs or carriage returns, which results in character-by-character analysis of the ends of the string, a painful and slow process in Visual Basic. The solution is to create a Visual Basic–native method function that does the trimming of white space for you, vbAllTrim. The code is shown below.

```
// AllTrim: Moves all control characters and spaces from both ends of
// a C string.
HLSTR __export _pascal vbAllTrim(LPSTR buffer)
{LPSTR lpStart=buffer,lpEnd;
 USHORT cb=0;
 HLSTR hlstr;
   while((unsigned char)*lpStart < 33)
      lpStart++; //Skip all characters that are spaces (x32) or below (x00.x32).
   lpEnd=lpStart;
   while(*lpEnd++ != 0); //Move to end of string.
   lpEnd--;             //Backtrack to first good character but don't overshoot.
```

```
    while((lpEnd > lpStart)  //Don't go before start.
        && ((unsigned char)*lpEnd < 33))
            lpEnd--;
    cb=1+(USHORT)( lpEnd-lpStart); //Bytes
    if (cb< 1) return NULL;
    hlstr=VBCreateTempHlstr(lpStart,cb);
    VALIDHLSTR(hlstr);
    return hlstr;
}
```

The use of unsigned char in the C code restricts characters deemed white space to values from 0 to 32, the control characters, and space. If a signed char was used, characters 128-255 would be included as white space because these characters are represented as negative numbers. This function obtains an LPSTR from Visual Basic, which means that the first Chr$(0) truncates the string. For homework, convert it to use a HLSTR parameter.

vbChangeChar

The vbChangeChar function returns a string with all occurrences of one character replaced by another character. We create new memory and use it to build the transformed string.

```
//--------------------------------------------------------------
// ChangeChar: Changes all of one character to another.
//--------------------------------------------------------------
HLSTR __export _pascal vbChangeChar(LPSTR buffer, LPSTR lpFrom, LPSTR lpTo)
{   //Create a Visual Basic string and copy.
    LPSTR lpStart;
    short cbBuffer= strlen(buffer) ;
    short I=0;
    HLSTR hlstr=VBCreateTempHlstr(buffer,cbBuffer);
    lpStart=VBDerefHlstr(hlstr);

    VALIDHLSTR(hlstr);  //Make sure of success.
 while(I < cbBuffer)   //Are we at the end of the string?
     {
     if(*lpStart==*lpFrom)  //If this character is to be changed, then
        *lpStart=*lpTo;     //change it.
         lpStart++;         // Move to next character.
       I++;
     }

    return hlstr;
}
```

The use of this function from Visual Basic is simple and is done with the following line that normalizes a file's UNC name:

```
    NormalUNC$=vbChangeChar(FileName$,"/","\")
```

For homework, convert this function to use an HLSTR parameter and do string substitution instead of character substitution.

Summary

This article shows how you can speed the coding of your Visual Basic and Microsoft Access code several fold, as well as improve its performance. The benefit of this method is having fewer lines of code to write, more robust exception handling, and a bit of C coding that you have to do only once. A further benefit is better isolation of the Visual Basic programmer from the operating system environment.

One of the strengths of Visual Basic is its string manipulations. The original Visual Basic functions can be augmented with additional functions using the Visual Basic–native methodology described here. Future technical articles will include some of my string libraries and discuss issues in their design.

One side benefit of this article is the ability to accurately time the performance of different coding styles. For many

Visual Basic users, changing coding style may increase the performance speed of their existing applications by 50 percent. A future technical article will examine performance issues in detail.

Bibliography

Appleman, Daniel. "Ten Commandments for Accessing the Windows API." Visual Basic Programmer's Journal, August/September, 1993.

Barlow, Chris, and Ken Henderson. "Mix C and VB for Maximum Performance and Productivity." Visual Basic Programmer's Journal, August/September, 1993.

Gunderson, Bob. "Extending Visual Basic with Microsoft Windows DLLs." January 1993.

Knowledge Base Q71106. "How to Pass One-Byte Parameters from VB to DLL Routines."

Knowledge Base Q112673. "How to Pass & Return Unsigned Integers to DLLs from VB."

Knowledge Base Q85108. "VB 'Bad DLL Calling Convention' Means Stack Frame Mismatch."

Lassesen, Ken. "A Collection of Useful Native Visual Basic and Microsoft Access Functions." August 1994. (MSDN Library Archive, Technical Articles)

The Cobb Group. "Accessing Initialization Files." Inside Visual Basic, July 1992 (Periodicals).

Visual Basic 3.0 Professional Edition Control Development Guide. Microsoft Corporation, 1993.