Tips for smooth migration of calls to Windows API methods

VB Migration Partner does a superb in dealing with Windows API calls. Here's a summary of the features that it supports:

- converts As Any parameters, by creating all the necessary overloads
- deals correctly with API methods that take a callback address (e.g. EnumWindows, EnumFonts)
- provides recommendation about the .NET object/method that can effectively replace the API method; we cover 300+ different API calls.
- ensures that string immutability doesn't prevent the .NET code from working correctly (see this article)
- generates the correct MarshalAs attributes for elements in Type (Structure) blocks
- correctly translates fixed-length strings inside Type blocks, so that they work correctly when passed to the Windows API method
- automatically initializes static arrays inside Type blocks, so that you don't get unexpected crashes when invoking an API method that expects to find a buffer there
- creates a wrapper method that ensures that orphaned delegates don't cause an unexpected runtime exception, an advanced programming technique discussed in this KB article
- includes the VB6WindowsSubclasser class that helps you correctly migrate subclassing-based techniques (as explained here)

In spite of all these features, there are cases when you still need to manually edit either the original VB6 code or the converted VB.NET. This happens, for example, if the original code uses the **VarPtr**, **StrPtr**, or **ObjPtr** functions to pass memory pointers to an external API method. These three functions aren't supported under VB.NET or C# and there is no simple way to simulate them. (Tip: You can use the VB6 Bulk Analyzer (available here) to quickly check whether your VB6 application contains the these functions.)

Tips for smooth migration of calls to Windows API methods

The good news is, in the vast majority of cases you don't need to deal with memory pointers under .NET, because the .NET Framework offer a valid "pure" alternative to the API method in question. This article illustrates a few steps that you might take to correctly migrate calls to Windows API – or to other external DLLs – and how you can take advantage of VB Migration Partner features to reduce manual edits to the very minimum and take advantage of the convert-test-fix methodology.

Avoid Windows API calls, if possible

You should always attempt to reduce direct calls to Windows API methods whenever it's possible to do so. Our experience is that many VB6 developers "love" to use Windows API methods directly, even if the VB6 language provides an alternative approach, because these methods usually perform faster. A typical example is the ubiquitous RtlMoveMemory method, often aliased as CopyMemory, which allows to perform super-efficient string and array operations. For example, consider the following VB6 code:

```
Declare Sub CopyMemory Lib "kernel32" Alias "RtlMoveMemory" _
        (dest As Any, dest As Any, ByVal numBytes As Long)


Sub FastArrayCopy(source() As Long, dest() As Long)
    ' we assume that both array have same size
    CopyMemory dest(0), source(0), 4 * (UBound(dest) + 1)
```

Tips for smooth migration of calls to Windows API methods

```
End Sub
```

This code works correctly even after the migration to .NET, but it would introduce a dependency from unmanaged code that can be avoided by rewriting the code as follows:

```
Sub FastArrayCopy(source() As Long, dest() As Long)

    ' we assume that both array have same size

    Dim i As Long

    For i = 0 To UBound(dest)

        dest(i) = source(i)

    Next

End Sub
```

You can also use pragmas to have the best of both worlds – the fast CopyMemory method under VB6 and the fully native code under .NET:

```
Sub FastArrayCopy(source() As Long, dest() As Long)

    ' we assume that both array have same size

    '## ReplaceStatement Array.Copy(source, dest, dest.Length)

    CopyMemory dest(0), source(0), 4 * (UBound(dest) + 1)

End Sub
```

Avoid undocumented VB6 function, if possible

If you can't avoid a call to an external DLL, at least try to not use the VarPtr, StrPtr, and ObjPtr method. For example, consider the following VB6 code, which uses a differente syntax for the RtlMoveMemory Windows API method:

```
Declare Sub CopyMemory Lib "kernel32" Alias "RtlMoveMemory" _
   (ByVal destAddress As Long, ByVal destAddress As Long, ByVal numBytes As Long)


Sub FastArrayCopy(source() As Long, dest() As Long)
    ' we assume that both array have same size
    CopyMemory ByVal VarPtr(dest(0)), ByVal VarPtr(source(0)), 4 * (UBound(dest) + 1)
End Sub


Function PeekWord(ByVal address As Long) As Integer
    ' read a 16-bit integer from memory
    Dim res As Integer
    CopyMemory address, ByVal VarPtr(res), 2
```

Tips for smooth migration of calls to Windows API methods

```
    PeekWord = res
End Function
```

In this particular case, the VarPtr method is used only because the RtlMoveMemory method expects a 32-bit address. You can rewrite the VB6 code so that no VarPtr method is necessary any longer, as follows:

```
Declare Sub CopyMemory Lib "kernel32" Alias "RtlMoveMemory" _
        (dest As Any, dest As Any, ByVal numBytes As Long)
Declare Sub CopyMemoryFromAddress Lib "kernel32" Alias "RtlMoveMemory" _
        (ByVal destAddress As Long, dest As Any, ByVal numBytes As Long)

Sub FastArrayCopy(source() As Long, dest() As Long)
    ' we assume that both array have same size
    CopyMemory dest(0), source(0), 4 * (UBound(dest) + 1)
End Sub

Function PeekWord(ByVal address As Long) As Integer
    ' read a 16-bit integer from memory
```

```
    Dim res As Integer
    CopyMemoryFromAddress address, res, 2
    PeekWord = res
End Function
```

Unfortunately, this trick isn't always applicable. For example, if you are passing a User Defined Type to a Windows API method and if a field in the UDT is expected to contain the address of another variable or structure, then you can't do without a VarPtr method. In such a case, you must use another approach, such as the one described next.

Use wrapper methods

If you can't avoid a call to a Windows API method, at least you should always wrap these calls in a method. By doing so, you can later replace those calls with references to "pure" .NET Framework objects and methods.

For simplicity's sake, let's focus on one of the simplest API methods, the GetSystemDirectory Windows API method. Here's a piece of VB6 code that displays the system directory path:

```
' Main.Bas module
Public Declare Function GetSystemDirectory Lib "kernel32.dll" Alias _
    "GetSystemDirectoryA" (ByVal lpBuffer As String, ByVal nSize As Long) As Long
```

Tips for smooth migration of calls to Windows API methods

```
Sub Main()

    Dim buffer As String, length As Long, windir As String

    buffer = Space(256)

    length = GetSystemDirectory(buffer, Len(buffer))

    winDir = Left(buffer, length)

    MsgBox winDir

End Sub
```

The first step is to refactor this code so that you make all the Declares private and move them to another BAS module, that exposes them by means of standard VB6 methods. (If you usually write tidy and maintainable VB6 code, odds are that you have already taken this step.)

```
' This is the APIHelpers.Bas file

Private Declare Function GetSystemDirectory Lib "kernel32.dll" Alias _

    "GetSystemDirectoryA" (ByVal lpBuffer As String, ByVal nSize As Long) As Long



' returns the Windows directory

Public Function SystemDirectory() As String
```

```
    Dim buffer As String, length As Long
    buffer = Space(256)
    length = GetSystemDirectory(buffer, Len(buffer))
    SystemDirectory = Left(buffer, length)
End Function
```

The code that actually displays or otherwise uses the Windows directory path is now simpler. Notice that we explicitly include the module name (APIHelpers) in the method call. This tip reduces the odds that another method with same name exists elsewhere in the project, but the technique explained later works even if you don't include such a prefix:

```
' the Main.Bas module
Sub Main()
    Dim windir As String
    winDir = APIHelpers.SystemDirectory
    MsgBox winDir
End Sub
```

At this point, you have a VB6 project that works exactly like the original one, but it is better organized and structured, with all Declares statements gathered in one single module. Let's see how to migrate this code to VB.NET and get rid of all dependencies from non-NET code.

Tips for smooth migration of calls to Windows API methods

First, we prepare a VB.NET module that exposes the same methods as the original APIHelpers.bas but doesn't use any Declare statement. Here's how we can render the SystemDirectory function using native .NET calls:

```
' This is the APIHelpers.vb file (VB.NET)

Module APIHelpers

    Public Function SystemDirectory() As String

        Return Environment.SystemDirectory

    End Function

End Module
```

Next, we use an ExcludeCurrentFile pragma to exclude the APIHelpers.bas VB6 module from migration process and we use an AddSourceFile pragma to add the APIHelpers.vb VB.NET file to the converted Visual Studio project. The neat result is that the code in Main now calls the .NET version of the method, which doesn't use any unmanaged calls:

```
' This is the APIHelpers.Bas file

'## ExcludeCurrentFile

'## AddSourceFile "c:\vbnet\modules\apihelpers.vb"


Private Declare Function GetSystemDirectory Lib "kernel32.dll" Alias _
```

Tips for smooth migration of calls to Windows API methods

```
    "GetSystemDirectoryA" (ByVal lpBuffer As String, ByVal nSize As Long) As Long

    ' ... remainder of module as before...
```

This solution works great, but we can improve it. In fact, a (minor) problem is that the resulting VB.NET code still uses wrapper methods and doesn't look like the "native" .NET code that an experienced VB.NET developer would write. Fear not, because all you need is a project-level PostProcess pragma:

```
' This is the APIHelpers.Bas file

'## ExcludeCurrentFile

'## project:PostProcess "(APIHelpers\.)?SystemDirectory", "Environment.SystemDirectory"
```

Notice that the AddSourceFile pragma has been dropped because you don't need the wrapper method any longer (at least in this simplified example). Using similar techniques you can provide a .NET equivalent for most methods that require API calls under VB6, including methods that take arguments.

One of the long-terms goals we have in Code Architects is to apply these concepts on a larger scale to create VB6 helper modules and their corresponding VB.NET versions, to help all VB6 developers to easily migrate their API-intensive applications.