What is the *VB Callback Server* Dll?

The Visual Basic Callback Server Dll allows you to use methods in a VB class module as callbacks for standard Windows API calls. Exposing callback entry points in Visual Basic allows you to easily use the EnumWindows, EnumFonts, and other windows enumeration functions without writing a C Dll to handle the function pointers required for these API calls. Other applications include timers without forms, controlling a dialog resource from VB, and installing Windows hooks. VBA is not thread-safe, so callbacks for threading aren't supported.
Since window procedures are considered callback functions, the callback server can also be used to subclass windows (in the same process, cross-process subclassing isn't supported). You can also use the Message Blaster OCX for subclassing. While MsgBlast is easier to use for the programmer making an early attempt at directly manipulating windows message and supports cross-process subclassing, the callback server has much less overhead. MsgBlast doesn't support callbacks, just subclassing of a window handle.

Setup?

To install the callback server, copy CBack32.Dll to the Windows\System directory (or System32 on Windows NT) and run regsvr32 CBack32.Dll. Regsvr32 can be found in the Tools\RegUtils directory on the VB5 CD. If you will be using the callback server to subclass and will be using break mode in VB, you should also copy VBBrk32.Dll to the system[32] directory. VBBrk32.Dll isn't required (in fact, it will not load) for a VB executable. VBBrk32 failing to load will cause Regsvr32 to post an error message, but you won't get this error when using CBack32 from a VB executable. If you want to insure CBack32 is properly installed on your client machine, add the following code to your program (assuming you have a reference to *VB CallBack Server*).

```
Private Declare Function DllRegisterServer Lib "CBack32.Dll" () As
Long
Sub Main()
Dim CBackGen As CallBackGenerator
    On Error Resume Next
    Set CBackGen = New CallBackGenerator
    If Err Then
        DllRegisterServer
        Err.Clear
        Set CBackGen = New CallBackGenerator
    End If
    If Err Then MsgBox "This program requires CBack32.Dll": End
End Sub
```

How do I use it?

To help you understand how to create a callback class module, we'll step through creating a class module to provide an EnumWindowsProc callback for the EnumWindows API call. We'll go over the details of each of the calls after you've seen the initial steps:

1. Add a reference to the *Callback Server* to your project.
2. Insert a class module into your project.
3. In the object browser, look at the *VBCallBackType* enumeration in the *VBCallBack* type library.
4. Select *CBType_WNDENUMPROC* and look at the description of the required method. You'll see *(ByVal Long, Any) As Long*. This information, together with the windows API help file for EnumWindows, will help you create a correct function prototype.
5. Make sure your editor is in *Full Module View.* You can change this setting using the Editor tab of the Options dialog.
6. Type *Public Function WNDENUMPROC(ByVal hWnd As Long, LParam As ListBox) As Long* into your class module. (The last parameter is user defined, we'll pass in a listbox just because we can).
7. Before the first method, type *Private m_CallBack As CallBack*
8. Add a Class_Initialize event. In the Class_Initialize event, type *Set m_CallBack = NewCallBack(CBType_WNDENUMPROC, Me, True)*
9. Add a read-only ProcAddress property as follows:

```
Public Property Get ProcAddress() As Long
```

```
    ProcAddress = m_CallBack.ProcAddress
End Property
```

You're now ready to create an instance of your class. After you create an instance of the class, you can pass the ProcAddress property of your class to the EnumWindows API call, and EnumWindows will call your WNDENUMPROC method.

What is the first method of a class?

The first method of a class is the first public entry in a class module. The callback server will only work if the first method is correctly defined. Public variables aren't allowed before the first method. To make sure the method you're defining is listed back first, be sure to use Full Module View. If you screw this up, you'll crash.

VBHandler

The VBHandler property of the CallBack class is used to set the VB class instance which will be used to handle the callback.

ProcAddress

The ProcAddress property returns the address you should use as the function pointer for your VB class.

Persistent ProcAddress, IsPersistent

Some callback addresses are used by a single API call. For example, you only have to count on the address of an enum procedure for the duration of the enumeration call. Other addresses, however, must remain constant for an indefinite amount of time. For example, a WindowProc stays the same as long as the window remains alive and is a persistent callback type. You can use the IsPersistent property of a callback object to see if the VBHandler class you assigned will always be called for the given procaddress. If a callback is persistent, then you can only create 32 instances of the callback object (32 was just as easy as 16).

ResetCallBack

If a callback address isn't persistent, then the same address is used for each instance of a callback with a given type. This means that the class instance which is being pointed to by the internal callback routine may be invalid if you have more than one instance of the same non-persistent callback type. Reading the ProcAddress property or calling the ResetCallBack method will guarantee that the next call will be mapped to the correct class instance. For example, if you are enumerating top-level windows and se a second class to enumerate child windows, you will need to do an m_CallBack.ResetCallBack before you return from the outer EnumWindows call. If you don't, then the next call made by EnumWindows will actually call the wrong enumerator class. In the sample code, this isn't an issue because the same enumerator is used recursively for top-level and child windows.

NewCallBack

The NewCallBack method of the CallBackGenerator class is used to create a callback object. You can't use the New keyword directly to create a callback object. The first parameter, which is required, specifies the type of callback object to be created. Note that NewCallBack will fail if you ask for more than 32 instances or a persistent callback type. The valid values for the Type parameter are listed in the VBCallBackType enumeration. If the callback type you use doesn't match your first method, you'll crash.

The VBHandler optional parameter is used as a shortcut for setting the VBHandler property. Pass in a reference to your class module.

The Contained optional parameter, which takes a boolean value, is a little harder to understand. If Contained is true, then NewCallBack does not increment the reference count (ie, AddRef), the object specified by the VBHandler parameter. This enables you to create a class module which uses a callback object and can be destroyed with a Set obj=Nothing call. If you don't use the contained flag, a Quit (or similar method) is required to clean up the object correctly. If you try to persist the callback object beyond the lifetime of the initial VBHandler, you must reset VBHandler. If you don't, you will crash. Normally, the call to NewCallBack is made in the Class_Initialize event and looks like:

```
Set m_CallBack = NewCallBack(CBType_WNDPROC, Me, True)
```

How does it work?

The concept behind the callback server is very simple. Visual Basic classes are interfaces which are derived from IDispatch. A VB class has a VTable, and you can generate a C++ header file which describes the VTable. Given an instance of a VB class, you can call its methods from C++ using the VTable. For example, the class for the WNDPROC looks something like:

```
class CVBWNDPROC : IDispatch
{
   STDMETHOD(CB_WNDPROC)(HWND hwnd, UINT msg, WPARAM wParam, LPARAM
lParam, LRESULT FAR*) = 0;
};
```

Reliance on the VTable is the reason why correctly defining the first method of the VB class is so important. The callback server doesn't care what the remaining methods and properties are, so long as the first method matches the class.

In order to have a callback, you must have a procedure address. Since member functions of classes can't be used reliably as function pointers (the implicit this pointer screws things up), the callback Dll has actual entry points for each defined callback type. The addresses of these fixed entry points are returned to the ProcAddress property of the callback class. These procedures simply call through the VTable into the VB class. For persistent callback types, there are 32 such entry points. Lets look at the first one:

```
LRESULT WINAPI CB_WNDPROC0 (HWND hwnd,
                            UINT msg,
                            WPARAM wParam,
                            LPARAM lParam)
{
  LRESULT retVal = 0;
  //Break mode checks, see next section
  if (*VBInBreakMode && VBInBreakMode() ) {
    if ( pCWNDPROC[0]->m_pDebugProc )
      return (pCWNDPROC[0]->m_pDebugProc)(hwnd, msg, wParam, lParam);
    else
      return 0;
  }
  //The real call.  pCWNDPROC is a global array of pointers to
  //CallBack classes.
  if (pCWNDPROC[0]->m_pVBClass)
      ((CVBWNDPROC FAR*) (pCWNDPROC[0]->m_pVBClass))->
       CB_WNDPROC (hwnd, msg, wParam, lParam, &retVal);
  return retVal;
}
```

What happens in break mode?

Visual Basic doesn't mind it when you call directly into VTables, unless it happens to be in break mode. If VB is in break mode, it doesn't expect any VB code to be executed. Calling the VTable directly bypasses all safety checks to stop code from executing, so the callback server must check this for you. The functionality for this is provided in a separate Dll called VBBrk32.Dll. This Dll exposes one entry point, called VBInBreakMode, which returns a BOOL. The Dll will fail to load if the callback server isn't running in the Visual Basic design environment. You can get a pointer to VBInBreakMode using LoadLibrary and GetProcAddress calls. By declaring a variable as follows *BOOL (WINAPI *VBInBreakMode)(void) = NULL;*, you can use the code shown above to determine if VB is in break mode or not.

DebugProc

So what happens when VB is in break mode? The answer is simple: the VTable call is disallowed. If the callback is being used to subclass, this causes a major problem because no windows message get through. To work around this, the callback class has a DebugProc property which can be set to the function pointer which is called in place of the VTable call when VB is in break mode. Search the sample code for DebugProc to see this in action.

Debugging Crashes

Since you are making VTable calls, you have to be very careful that your VTable and parameters are set up correctly. If you are experiencing crashes when your callback method is called, try to put a break point on the first (ie, the Sub or Function line) of the callback method. If you don't reach the breakpoint before crashing, then you've misplaced the first method. Be sure there are no public methods, properties, or variables listed before your callback method. If you reach the break point, but crash if you step off of it, then your parameters are wrong.
If you are running the 16 bit callback server, you may run into conflicts with VB when the Appearance property of controls on a subclassed form is set to *1 - 3D*. This isn't a problem for the 32 bit server.

Workarounds for parameter type limitations

VB classes can't take structures as parameters. This causes some problems for some callbacks, which pass in structures. When your callback takes a structure, define the parameter as a ByVal Long and use CopyMemory to make a copy of the structure. See WM_GETMINMAXINFO or EnumFontFamProc in the sample project.
Another problem occurs with strings. VB strings are actually BSTRs, but callback strings are passed in as LPSTRs. You can use CopyMemory to make a copy of the string. See the ResourceEnumerator class in the sample to see an example.

How to build the callback server

The code for the callback server is included. Everything is highly macroized, so you need to modify only the CBackDat.h file to add or remove callbacks. Beneath the main directory, there are IP32 and IP16 (IP=In-process, I use this for all of my servers). The IP directories contain the VC mak files. In order to build the project, you first need to build the TLB and header files using TLB.Bat, which should be called from the IP32 directory with ..\TLB 32 H before you attempt to build the 32 bit callback server.
Although I've tried to include as many callbacks in the server as I could find, you may find some that I missed. You may also notice that the Dll could be smaller because I have many more callbacks defined than you can ever use in a single project. If you wish to build a custom version of the callback server, feel free to do so. However, please call the server something else, rename the file, and change the GUIDs in CBack.ODL before redistribution of your server.
The code for the VBBrk32 project isn't included. This Dll will need to be replaced for future version of Visual Basic (hence the generic name). Any code you derive from reverse engineering this Dll is explicitly not supported. You can use it the same way I used it and get the same results.

Copyright information

CBack32.Dll and CBack16.Dll can be redistributed freely, as can derivations of the server. However, the sample code and rights to the callback server belong to Microsoft Corporation and are copyrighted with this book. Redistribution of the sample code is prohibited.

I hope you enjoy using my callback server. I'd love to hear your feedback, as well as feature requests for future versions of the callback server Dll. You can contact me at MattCur@Microsoft.Com.
Matthew Curland