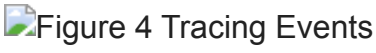
Cutting Edge: Windows Hooks in the .NET Framework



New information has been added to this article since publication.
Refer to the Editor's Update below.

Cutting Edge

Windows Hooks in the .NET Framework

Dino Esposito

Code download available at:CuttingEdge0210.exe(156 KB)

Contents

If you wanted to, you could distinguish two general categories of classes in the Microsoft® .NET Framework—classes that introduce new functionality such as XML readers and ADO.NET providers, and collections and classes that wrap underlying Win32® system functions. However, the .NET Framework provides classes for only some of the basic Win32 system routines. For many other functions, it still relies a great deal on the native Win32 API. An example is the stream classes of System.IO. The stream classes (such as FileStream) perform their I/O operations using the Win32 CreateFile function. Likewise, the really cool FileSystemWatcher class is just an easy-to-use wrapper built around the ReadDirectoryChangesW API. (By the way, this explains why the class FileSystemWatcher can't be used on Windows® 98 and Windows Me clients; the ReadDirectoryChangesW API is supported only on Windows NT® 4.0 and later.)

Most Win32 functionality has a managed counterpart now, either natively provided or achieved through the P/Invoke interoperability layer. Among the various classes that expose managed wrappers for Win32 APIs, those that are lacking include NTFS functions, memory-

mapped files, serial port support, and Windows hooks. In this column, I'll tackle Windows hooks from the perspective of .NET applications and discuss how to build .NET Framework wrapper classes for them. Before continuing, I should point out that a managed C++ approach would be more straightforward, if your project allows for it.

A Win32 Hooks Refresher

In Win32, a hook is a mechanism by which a user-defined function can intercept a handful of system events before they reach an application. Typical events that hooks can intercept include messages, mouse and keyboard actions, and opening and closing of windows. The hook can act on events and modify the actual flow of code. A hook is ultimately a callback function that applications register with a particular system event.

The signature of the hook function—frequently referred to as the filter function—varies quite a bit according to the type of event it is expected to trap. The target event determines the signature, the content of the arguments, and the behavior of the filter function. When Windows is going to perform a task that might have a hook (like creating a new window), it looks at the collection of hooks for that event. If any hook function is found, the operating system yields to it and regains control only at the end of the hook chain.

[**Editor's Update - 5/12/2004:** The text originally stated that the function installed last would be at the end of the chain. This has been fixed.] Each hook function is responsible for passing control to the next one in the chain when finished. If the filter function doesn't call the next hook, the operating system takes control of the flow and considers the event-hooking phase completed.

Applications register functions to hook certain events using API functions like SetWindowsHookEx, which adds a filter function to a particular event's hook chain, and UnhookWindowsHookEx, which removes it.

A fundamental aspect of hooks is their scope. Normally, hooks may have either system or thread scope. A few, however, can only have system scope. When a hook works at the thread level, it can only trap events generated within that thread. For example, a keyboard hook gets invoked only for the keystrokes directed to the thread's input queue. Similarly, a systemwide mouse hook gets called whenever the user moves the mouse, regardless of the particular thread that handles the event. A system-scoped hook is called to handle the event for all the currently running threads. This poses a precise context problem. How can a piece of code defined in one Win32 process invade the memory space of another process? To allow for this, a systemwide hook must be defined in a DLL so that the system can easily inject that code into each of the Win32 process memory spaces.

Even from this brief description, it's clear that thread hooks (or local hooks) are patently more efficient than system hooks (or global hooks). On the other hand, they cannot accomplish all the tasks global hooks can. In this column, I'll focus on thread hooks.

In **Figure 1**, you can see the list of the seven most frequently used hooks that have been available since Windows 2000. Bear in mind that the list is in no way exhaustive; many more hooks are available and they're covered fully in the MSDN® documentation.

Figure 1 Frequently Used Hooks

Hook	Description
WH_CBT	Window-related events such as creation, destruction, activation, min/max
WH_KEYBOARD(_LL)	Keyboard-related events
WH_MOUSE(_LL)	Mouse-related events
WH_GETMESSAGE	Message extracted from the main loop
WH_MSGFILTER	Menu- or dialog-generated messages
WH_CALLWNDPROC	Before calling the wndproc
WH_SHELL	Shell events such as start/close apps

Process Separation in the .NET Framework

The .NET Framework provides no built-in facilities or infrastructure to handle hooks. Right now, hooks are considered merely a special breed of callback functions and their implementation is left to P/Invoke (the .NET Framework infrastructure to call unmanaged APIs residing in the underlying operating system).

Before going any further with practical examples of system hooking in .NET, let me point out a platform-related aspect that marks Win32 and the .NET Framework. In Win32, the smallest unit of processing is the process. The CPU works by allotting slices of time to each process. While running, a process sees the whole 32-bit range of memory at its disposal. For this reason, it's architecturally impossible for a process to inadvertently corrupt another process' memory. (A process can still break another process, but they must both be explicitly using globally shared memory.)

In the .NET Framework, there is a significant change to this process. The .NET managed code runs under the control of the common language runtime (CLR) module and is subject to inspection and verification before execution. The CLR enables a piece of managed code only if it can be marked as type-safe code. The verification process ascertains the correctness of the intermediate language (IL) code and ensures that it accesses only authorized memory locations. In addition, type-safe code is guaranteed to reference only strictly compatible types and call objects only through properly defined types.

Type safety is critical to assembly isolation and security. Thanks to type safety, the CLR can use a unit of processing more lightweight than processes to implement code isolation. In .NET, AppDomains rather than processes are the smallest unit of processing among which communication is ruled by the system. In light of this, it's foreseeable that .NET native hooks would provide for a third level of scope: current thread, all threads in the AppDomain, and even all threads in the system desktop. In other words, don't wait for the hooks to be implemented in the .NET Framework. It could take a while because of the nontrivial implications they'd have, due to the architectural changes introduced in the transition from Win32 to the .NET virtual machine.

Hooks in .NET

Setting up hooks in .NET-centric apps is as easy as calling the underlying API functions—that is, SetWindowsHookEx to install a hook and UnhookWindowsHookEx to uninstall. To issue calls to Win32 API functions from within .NET applications, you must first import the desired API declarations into some sort of managed class. The whole process is not significantly different from using the Win32 API in applications written with Visual Basic® 6.0.

In **Figure 2**, you can see the C# code to import all the Win32 definitions needed to handle hooks. Functions are imported as static, externally defined members of a new .NET class. Any .NET class can access the Win32Hook class and invoke members. When this happens, the P/Invoke infrastructure guarantees that the call is marshaled back and forth to the system API across the CLR. Let's focus on the declaration of the key functions:

```
[DllImport("user32.dll")] public static extern IntPtr SetWindowsHookEx(HookType code, (
```

Figure 2 Importing Win32 Definitions

```
// ************************************************************** // Win32: Se
```

The DllImport attribute determines the source DLL of the function. Bear in mind that in order to use this attribute you must import the System.Runtime.InteropServices namespace. For more information about the P/Invoke layer and the various options available while importing

Win32 functions, take a look at David Platt's article "Get Ready For Microcost .NET by Using Wrappers to Interact with COM-based Applications" in the August 2001 issue.

In Win32, the SetWindowsHookEx function has a slightly different prototype:

```
HHOOK SetWindowsHookEx(int code, HOOKPROC func, HINSTANCE hInstance, DWORD threadID);
```

Mapping types correctly is a crucial task for a safe importation of hook code in the managed environment. Any Win32 handle, including HWND, HINSTANCE, and HHOOK, should be imported as IntPtr, though you can often simply use an integer. Delegates are the managed counterpart to callback functions. In Win32, the HOOKPROC type defines a pointer to a generic callback function to which all filter functions of all hooks will conform:

```
LRESULT CALLBACK HookProc(int code, WPARAM w, LPARAM l)
```

A filter function returns an integer and takes three arguments. The first is an integer that indicates the action the hook is going to take. The second and third arguments are generic containers of hook-specific information. In Win32, they are passed as plain old 32-bit integers. In .NET, you can either use the IntPtr or the integer type. Since variables in .NET are strongly typed and not easily converted between types, I highly recommend that you use the IntPtr type. The reason is that with hooks you often need to cast that value to a particular data structure, a practice that is forbidden if the base type is a value type, as is an integer. Avoid using the object type, because the .NET object type is marshaled as an OLE Variant. An interoperability exception is raised if the actual data is not a Variant, which is just the case with hooks. The following code shows the delegate to render the HOOKPROC type:

```
public delegate int HookProc(int code, IntPtr wParam, IntPtr lParam);
```

An interesting aspect of the .NET interoperability with Win32 native code is that to some extent you can customize the types being used in managed code. Of course, this sort of customization is possible as long as you maintain the size of the elements being pushed on the stack. For example, you can easily replace a rather generic integer type with values from an integer enumeration. Although it's declared as an integer, the code argument of the SetWindowsHookEx function can actually take only a few well-known values:

```
static extern IntPtr SetWindowsHookEx(HookType code, HookProc func, IntPtr hInstance, :
```

If you create an enum type that accepts only the proper values, you can replace the int type with the enum, resulting in more maintainable and readable code that fully integrates with Visual Studio® .NET.

A Class for Windows Hooks

Since the .NET Framework is composed of a hierarchy of classes, the best way to import Win32 hooks in the .NET Framework is through a class. Designing such a class is greatly simplified because a common pattern rules the behavior of all hooks. Despite the fact that you can choose among a dozen different hooks, most of what they accomplish can be described by the same set of instructions. This is an ideal situation for creating a base class—the LocalWindowsHook class. As I mentioned, in this column I'll focus on local hooks. However, aside from the different kinds of impact they have on the system, local and system hooks differ only by a couple of arguments in the call to SetWindowsHookEx.

The entire source code for the LocalWindowsHook class can be seen in **Figure 3**. The class has a couple of constructors. To start out, you need to specify the type of hook you want to create—a value picked up from the HookType enumeration. In addition, you can either rely on the default filter function or provide your own:

```
public LocalWindowsHook(HookType hook) public LocalWindowsHook(HookType hook, HookProc
```

Figure 3 LocalWindowsHook Class

```
using System; using System.Runtime.InteropServices; namespace MsdnMag { public class H
```

The filter function of a hook must observe a few rules, like calling CallNextHookEx to yield to the next hook in the chain:

```
protected int CoreHookProc(int code, IntPtr wParam, IntPtr lParam) { if (code < 0) retu
```

The m_hhook member is an internal property that tracks the handle of the created hook. The property is set with the return value of SetWindowsHookEx. You install a hook by calling the Install method and you remove it from the chain with a call to Uninstall. Internally, the Uninstall method calls UnhookWindowsHookEx and passes the m_hhook handle on to it:

```
public void Install() { m_hhook = SetWindowsHookEx( m_hookType, m_filterFunc, IntPtr.Z
```

SetWindowsHookEx accepts previously stored hook type and filter function reference. The third argument should be the HINSTANCE handle of the DLL that contains the code for the filter function. Typically, this value is set to NULL for local hooks. Do not use the .NET null object, though; use the IntPtr.Zero expression, which is the correct counterpart for Win32 null handles. The HINSTANCE argument cannot be null for systemwide hooks, but must relate to the module that contains the hook code—typically an assembly. The Marshal.GetHINSTANCE static method will return the HINSTANCE for the specified .NET module.

The fourth argument of SetWindowsHookEx is the key to determining whether the hook should work locally or globally. The argument denotes the ID of the thread affected by the hook. If this argument is zero, then all the currently running threads in the current system

desktop will be affected. Implementation of hooks specific to .NET could improve on this by adding a third option that would affect all the threads in the current AppDomain.

Local hooks set the thread ID argument of SetWindowsHookEx with the ID of the current thread. The AppDomain.GetCurrentThreadId method returns the needed value. The AppDomain's method is a simple wrapper built around the Win32 GetCurrentThreadId function.

The most interesting part of the hook implementation takes place in the body of the filter function. Each hook type packs different data in the wParam and lParam arguments of the filter function. In addition, the feasible actions are different from one hook to the next. You need code that adheres to the specification of the particular hook type, but this code may vary quite a bit in distinct hook instances. Therefore, I made the LocalWindowsHook class fire an event so that derived classes can easily inject the code that fits their needs.

```
public event HookEventHandler HookInvoked;
```

The HookInvoked event is raised by the CoreHookProc method. To allow for effective processing of the hook data, the event must carry on the client all the information available to the filter function. For this reason, you need a custom event handler (HookEventHandler) and a custom event data class (HookEventArgs). In particular, the event data class inherits from EventArgs and extends it with three properties—HookCode, wParam, and lParam—that are containers for the arguments that Windows passes on to the filter function. The following code snippet shows how the hook class fires the event:

```
HookEventArgs e = new HookEventArgs(); e.HookCode = code; e.wParam = wParam; e.lParam =
```

Before discussing how to build specific hook classes, let me share a final note on the LocalWindowsHook class. In order to call into Win32 API functions, you simply need a class to which you add a static method. This does not have to be a new class. The LocalWindowsHook class, in fact, exposes the functions SetWindowsHookEx, UnhookWindowsHookEx, and CallNextHookEx as protected methods. In this way, they are ready to use for derived classes that need to replace the filter function altogether.

Building a CBT Hook Class

Although the LocalWindowsHook class is not abstract, it is hardly usable in code. When you need a hook, the best you can do is create a derived class, set the hook type, and handle the HookInvoked event as appropriate. Let's see how to proceed for a WH_CBT hook. In spite of the rather misleading name, this hook has a lot to do with window events. For example, a WH_CBT hook lets you intercept the creation and the destruction of any type of window. In a certain sense, a WH_CBT hook fires events whenever a call to CreateWindow or

DestroyWindow occurs. The sample hook that I'll build shortly traces creation, destruction, and activation events that are triggered by a call made to MessageBox.Show.

Let's start at the end and take a look at the final code that ultimately makes the hook run:

```
// the CbtHook is already set up void MsgBox_Click(object sender, EventArgs e) { cbt.I
```

This code produces a message box containing a dialog window, text, and a button (see **Figure 4**). When windows are created, their title is not set. This is due to the internal implementation of the MessageBox API function, which is still the underlying API that outfits the MessageBox.Show method. When the dialog box is activated, the title has been correctly set to "Courtesy of Cutting Edge," as in the sample code. In the top of **Figure 4** you see that the tracing occurred until then. When the dialog box is dismissed, the parent form is activated and then the dialog is destroyed.

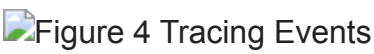


Figure 4** Tracing Events **

At this point, you're probably wondering if you should create a derived class for each particular hook of a certain type that you need to build. Well, not exactly. Two distinct classes contribute to the output of **Figure 4**—the generic LocalCbtHook hook class, which provides the intercepting infrastructure, and the Windows Forms application, which implements tracing. Let's see how.

The LocalCbtHook class inherits from LocalWindowsHook and defines its constructors as follows:

```
public LocalCbtHook() : base(HookType.WH_CBT) { this.HookInvoked += new HookEventHandl
```

Both constructors call the respective parent constructor specifying a particular hook type—HookType.WH_CBT. After that, constructors register a handler for the HookInvoked event. This internal event handler is responsible for adapting the base filter function to the needs of the particular hook type (see **Figure 5**).

Figure 5 Implementing Hooks

```
private void CbtHookInvoked(object sender, HookEventArgs e) { CbtHookAction code = (Cb
```

The CbtHookInvoked method translates the code argument from an integer type to a member of the more specific CbtHookAction enum type. I built the enum by looking at the range of possible values defined in Win32 for the code argument of the WH_CBT hook filter function.

Typical actions for a WH_CBT hook include the creation and the destruction of a window, activation, and the change of focus, size, and position. The sample LocalCbtHook class provided with this column handles only three actions: HCBT_CREATEWND, HCBT_DESTROYWND, and HCBT_ACTIVATE. For each of them, the class fires an event so that the caller can easily execute application-specific tasks: WindowCreated, WindowActivated, WindowDestroyed. Also in this case, I used a custom event handler (CbtEventHandler) and an event data structure (CbtEventArgs). Events fire from the HandleXxxEvent routines called by the filter function (see **Figure 5**).

The custom event data exposes some window-specific information such as the Win32 handle, the class name, and the title. In addition, a Boolean flag is set to indicate whether the window being created is truly a window or a dialog:

```
public class CbtEventArgs : EventArgs { public IntPtr Handle; public string Title; pub
```

The information needed to fill these fields is retrieved using other Win32 API functions—in particular, GetClassName and GetWindowText. These methods are imported as protected static members in the LocalCbtHook class.

Using the CBT Hook

A client application, like the one shown in **Figure 4**, sets up the hook as shown in the following code snippet:

```
cbt = new LocalCbtHook(); cbt.WindowCreated += new CbtEventHandler(WndCreated); cbt.Wi
```

The action taken when a particular hook-specific event occurs is defined in the handler of each event. The sample application in **Figure 4** simply traces the events as they arrive:

```
public void WndCreated(object sender, CbtEventArgs e) { string buf = "Created {0}, cla
```

I designed the LocalWindowsHook and the LocalCbtHook classes to be as generic as possible, but also to limit the amount of boilerplate code. Thanks to the object-oriented nature of the .NET Framework and the event mechanisms present, you can set up hooks in a very natural way without knowing the underlying API. Creating new hook classes from LocalWindowsHook is rather straightforward. You just derive the new class, specifying the hook type you want and then handle the HookInvoked event to do what the specific hook requires you to do.

All the key source code available with this article is written in C# but, again, thanks to the language neutrality of the .NET Framework you can easily use Visual Basic .NET to build new hook classes or consumer applications. The sample application is also available in Visual Basic .NET.

Send questions and comments for Dino to cutting@microsoft.com.

Dino Espositois an instructor and consultant based in Rome, Italy. He is the author of *Building Web Solutions with ASP.NET and ADO.NET* and the upcoming *Applied XML Programming for Microsoft .NET*, both from Microsoft Press. Reach Dino at dinoe@wintellect.com.