Trapping windows messages in the Microsoft® .NET Framework

Introduction

To perform some tasks, we still need to trap unmanaged windows messages, even though we are developing managed code. This is a little tutorial that tries to clarify how to trap those messages using the .NET Framework.

The code is written in C#. Even though I believe it will be quite easy for those who are reading this document, but use VB.NET™ instead of C#, to port the concepts.

Learn by example

Now, to clarify the concepts lets use something useful. Maybe some of the readers have the need to detect if a CD, or for what matters any removable volume mounted on a device which supports a software ejection method (DVD, Zip, etc…), has been inserted or removed from a device. This can be accomplished by detecting the *WM_CHANGEDEVICE* message.

Trapping messages in .NET

There are, to my knowledge, two ways of trapping windows messages in the Microsoft® .NET Framework.:

The IMessageFilter interface

The most obvious, and I suspect the least useful, is to use the *IMessageFilter* interface.

The CLR defines an *IMessageFilter* interface, which describes a single method named *PreFilterMessage*, which can be used to trap Windows messages. After defining a class which implements the *IMessageFilter* interface, we just need to tell the Application class, that it should add it to the queue of *IMessageFilter* interfaces it can handle using the *Application.AddMessageFilter* method. We can also remove a filter using the *Application.RemoveMessageFilter* method from the queue. Now, this approach is only useful to trap dispatched messages, it does not handle all messages. This process is out of scope in this document, anyway, as an example example, we would have to use something like:

```
using System;
using System.Windows.Forms;

public class MyFilter: IMessageFilter
{
      public bool PreFilterMessage(ref Message aMessage)
      {
            if (aMessage.Msg==WM_AMESSAGE)
            {
                  //WM_AMESSAGE Dispatched
                  //Let's do something here
                  //...
            }
            // This can be either true of false
            // false enables the message to propagate to all other
            // listeners
            return false;
      }
```

```
}
```

At some other point of your code you would have to register an instance of *MyFilter*, like:

```
// This will have to have a scope that makes it accessible to both
// AddFilter and RemoveFilter
MyFilter fFilter = new MyFilter();
(…)
Application.AddMessageFilter(fFilter);
(…)
Application.RemoveMessageFilter(fFilter);
```

Overriding a WndProc method

The solution, in our example, is overriding a *WndProc* method of a *Control* or to do the same with an implementation of the *NativeWindow* class. The first solution is useful if we need to trap messages in a class that inherits the Control class. If we want to build a class that is not dependent on a specific *Control*, we will have to use the *NativeWindow* class. This class simply wraps a window handle, and so, it enables us to override the *WndProc* method which is implemented.

In both cases, we will override the *WndProc* method like:

```
protected override void WndProc(ref Message aMessage)
{
      if (aMessage.Msg==WM_AMESSAGE)
      {
            //WM_AMESSAGE Dispatched
            //Let's do something here
            //...
      }
}
```

Detecting a volume insertion or removal

Now, let's look at our example. We already know that we will need to trap the *WM_DEVICECHANGED* message, and to filter two specific events. We will build the example based on two classes.

The first one, *_DeviceVolumeMonitor*, will be private to the project, and will only serve to inherit the *NativeWindow* class, so that our main class won't expose the public *NativeWindow* methods. This class will also serve the purpose of encapsulating all API constants and structures that we will need to make things work.

DeviceVolumeMonitor, will be our main class. It implements most of the logic needed to the user. Nevertheless, the main subject of this tutorial is in the other class.

The WM_CHANGEDEVICE message

This message is broadcasted whenever something relevant occurs in a device connected to Microsoft® Windows®. For this matter, there are two events which we will need to trap, the *DBT_DEVICEARRIVAL* and the *DBT_DEVICEREMOVECOMPLETE*.

Both events are handled the same way, the only difference being that *DBT_DEVICEARRIVAL* detects a volume being inserted, and the *DBT_DEVICEREMOVECOMPLETE* detects that a volume was removed. Now, we need to look at some of the *Win32* API declarations to see how this works.

The *WM_CHANGEDEVICE* constant definition can be found in *WinUser.h*, all other information we will need, it can be found in *DBT.h* header file. Both files are part of the Microsoft® Platform SDK.

When a *WM_CHANGEDEVICE* is broadcasted, the message structure will transport the event value in the *WParam*, and some additional data in a location pointed by *LParam*.

As I already described, we are only interested in trapping messages that have a *WParam* with the *DBT_DEVICEARRIVAL* or *DBT_DEVICEREMOVALCOMPLETE* values. The data pointed by *LParam* will have the same structure in both cases.

Once one of the events is detected, we must cast the pointer stored in *LParam*, to a *_DEV_BROADCAST_HDR* structure, and evaluate the *dbch_devicetype* field. We are only interested in proceeding if the field value is *DBT_DEVTYP_VOLUME*.

If this is the case, then we will need to cast *LParam* again, but this time to point at a *_DEV_BROADCAST_VOLUME* structure. This structure has two fields which we will have to evaluate. The first one is *dbcv_flags*. This field will tell us the nature of the volume that was mounted (or dismounted). It can be either a media volume like a CD (the flag will have a *DBTF_MEDIA* flag) or a network share (*DBTF_NET*). In this case we will only be interested in filtering the *DBTF_MEDIA* value. The other field is *dbcv_unitmask*, which is a bit vector that tells us which drive letters were mounted.

At this point, there are two facts which need to be clarified. As stated in the Platform SDK documentation, this message can tell us if more than one volume was mounted and these can be of multiple types. We will have to make an assumption that is whenever the *dbcv_flags* includes the *DBTF_MEDIA* value, we will assume that all the drives described by the *dbcv_unitmask* bit vector are of this type. I don't think that it is likely that this notification will ever have a *dbcv_flags* that includes both types. Anyway, this would be solved if we would determine the specific type for all the drives described by *dbcv_unitmask*. On the other hand, as it was already implied, *dbcv_unitmask* can describe more than one drive at a time. I guess this is only likely if we have a jukebox device, but anyway this is a problem we will solve in the code.

The _DeviceVolumeMonitor class

This is a helper class which is internal to the project. Now as I told you before, this class is only useful to inherit the *NativeWindow* class and to encapsulate the API constants and structures we will use. The class constructor takes only a parameter which will be the instance of the main class that owns the *_DeviceVolumeMonitor* instance.

The API constants and structures were modified to become more readable and useful in the .NET context.

The *WndProc* is overridden according to the cascading conditions described for the *WM_DEVICECHANGE* message. If all conditions are fulfilled then the object uses as internal method of the main class to inform it that a valid event has occurred. This method is called *TriggerEvents*.

The DeviceVolumeMonitor

Our main class, contains some logic, which has no relation to windows messages itself.

Now, we know that this class creates an instance of the former class, and that whenever an event is detected, it is reported back to the main class through the *TriggerEvents* method. This is not a clean OOP approach, but there was no point in making things more complicated.

This class has two properties defined: *Enabled* and *AsynchronousEvents*. The *Enabled* property is self explanatory, is handles enabling or disabling the message trapping. This is done with the helper class. When we want messages to be trapped we assign the handle to the *NativeWindow* descendant, and the *WndProc* method will be invoked whenever there is a message either broadcasted or dispatched, the handle will be released when the property is unset. This approach is better than controlling the property value in the *WndProc* method (the property is still evaluated on the method as a sanity check anyway), because the class won't overload the system with unnecessary checks.

AsynchronousEvents controls the way events are invoked. If it is set, then the event will be called asynchronously, and *WndProc* won't stall waiting for the application to process whatever it has to. Now, the way this is done on the *TriggerEvents* method is not peaceful. Many of you will criticize the *BeginInvoke* without following the design pattern. It is an unsafe option, but still it works if it only evolves calling thread safe code. Either way, I assume that you know what you are doing if you set this property.

There is a platform invoke to an API function called *QueryDosDevice*. This is a function that enables the translation between the drive letter, or DOS device name, to the full device path. This is used by one of two methods which translate the bit vector that was mentioned before. *MaskToLogicalPaths* translates the bit vector to a comma delimited string with all the drive letters described by the bit vector, *MaskToDevicePaths* on the other hand returns a string which is a list of the full device paths.

Two events are defined, *OnVolumeInserted* and *OnVolumeRemoved*. Both of them take the issued bit vector as a parameter.

Finally, I have implemented the *IDisposable* interface, I followed the design pattern, so there is not much to say here.

Rui Reis

April 2003