On the need to keep most event sources alive if you want them to raise events

devblogs.microsoft.com/oldnewthing/20231123-00

November 23, 2023

Raymond Chen

A customer was having difficulty registering for the Windows Runtime `Radio.StateChanged` which notifies you that the state of a radio (such as a cellular radio or WiFi radio) has changed. They were able to narrow it down to a small program:

```
#include <stdio.h>
#include <winrt/Windows.Devices.Radios.h>
#include <winrt/Windows.Foundation.h>
#include <winrt/Windows.Foundation.Collections.h>

using namespace winrt;
using namespace winrt::Windows::Devices::Radios;
using namespace winrt::Windows::Foundation;
using namespace winrt::Windows::Foundation::Collections;

void OnRadioStateChanged(Radio const& sender, IInspectable const& args)
{
    printf("Radio state changed: %ls\n", sender.Name().c_str());
}

void RegisterRadios()
{
    auto radios = Radio::GetRadiosAsync().get();
    for (auto radio : radios) {
        radio.StateChanged(&OnRadioStateChanged);
        printf("Registered event on radio %ls\n", radio.Name().c_str());
    }

}

int main()
{
    init_apartment();
    RegisterRadios();

    std::cout << "Waiting for radios to change\n";
    getchar();
}
```

They found that the program registered its events successfully, but the event handler was never called.

The problem is that they registered the events on the `Radio` objects that were returned by `GetRadiosAsync()`, but then allowed those references to `Radio` objects to destruct.

As a general rule, when a program releases its last reference to a Windows Runtime object, the Windows Runtime object destructs, and that means it won't generate any events.

There are exceptions to this principle. For example, there may be other references to the object being held by components outside the program itself. XAML elements, for example, remain alive when they are part of a visual tree because the XAML parent object has a reference to the XAML element child. And DispatcherTimer objects retain a reference to themselves as long as they are still ticking. Another category of objects that are kept alive externally are those of the form `T.GetForCurrentView()`, which are per-view singleton objects which remain valid as long as the view remains valid.

Now, in garbage-collected languages, the exact point that a reference is run down by the garbage collector is generally unpredictable. Consider the equivalent C# program:

```
using System;
using Windows.Devices.Radios;

class Program
{
    static void OnRadioStateChanged(Radio sender, object args)
    {
        Console.WriteLine($"Radio state changed: {sender.Name}");
    }

    static void RegisterRadios()
    {
        var radios = Radio.GetRadiosAsync().Result;
        foreach (var radio in radios) {
            radio.StateChanged += OnRadioStateChanged;
            Console.WriteLine($"Registered event on radio {radio.Name}");
        }
    }

    public static void Main()
    {
        RegisterRadios();

        Console.WriteLine($"Waiting for radios to change");
        Console.ReadLine();
    }
}
```

Those `Radio` objects that were obtained by the `RegisterRadios` method are going to be released by the garbage collector at some unspecific point in the future, and the effect on your program is going to be that it seems to be working for a while, and then suddenly stops receiving radio events.

You need to keep the `Radio` objects alive if you intend to receive events from them.

```
#include <stdio.h>
#include <winrt/Windows.Devices.Radios.h>
#include <winrt/Windows.Foundation.h>
#include <winrt/Windows.Foundation.Collections.h>

using namespace winrt;
using namespace winrt::Windows::Devices::Radios;
using namespace winrt::Windows::Foundation;
using namespace winrt::Windows::Foundation::Collections;

void OnRadioStateChanged(Radio const& sender, IInspectable const& args)
{
    printf("Radio state changed: %ls\n", sender.Name().c_str());
}

auto RegisterRadios()
{
    auto radios = Radio::GetRadiosAsync().get();
    for (auto radio : radios) {
        radio.StateChanged(&OnRadioStateChanged);
        printf("Registered event on radio %ls\n", radio.Name().c_str());
    }
    return radios;
}

int main()
{
    init_apartment();
    auto radios = RegisterRadios();

    std::cout << "Waiting for radios to change\n";
    getchar();
}
```

For C#:

```
using System;
using System.Collections.Generic;
using Windows.Devices.Radios;

class Program
{
    static void OnRadioStateChanged(Radio sender, object args)
    {
        Console.WriteLine($"Radio state changed: {sender.Name}");
    }

    static IList<Radio> RegisterRadios()
    {
        var radios = Radio.GetRadiosAsync().Result;
        foreach (var radio in radios) {
            radio.StateChanged += OnRadioStateChanged;
            Console.WriteLine($"Registered event on radio {radio.Name}");
        }
        return radios;
    }

    public static void Main()
    {
        var radios = RegisterRadios();

        Console.WriteLine($"Waiting for radios to change");
        Console.ReadLine();

        GC.KeepAlive(radios);
    }
}
```

It’s more common to keep the objects alive by saving them in a member variable of the same class that is handling the events, but in our examples here, we are using static methods rather than instance methods, so we don’t have that luxury. It’s either keep them in global variables (boo, global variables) or keep them in local variables whose lifetimes extend beyond the useful lifetimes of the event handlers.