Error 0x80131040 “The located assembly’s manifest definition does not match the assembly reference” when I use a Windows Runtime class written in C# from my C++/WinRT project

devblogs.microsoft.com/oldnewthing/20200615-00

June 15, 2020

Raymond Chen

So you’re writing your C++/WinRT project, and everything is going swimmingly. There’s some code written in C# that you’d rather not port to C++/WinRT, just use as-is. “No problem,” you say. “I can just package the C# code as a Windows Runtime class, and then I can use the inter-language features of the Windows Runtime to allow the C# code to be consumed by C++/WinRT.”

You add the C# code as a project alongside your C++/WinRT project, you add a reference from the C++/WinRT project to the C# project, and everything builds: It’s a miracle!

Except that when you actually try to use the C#-written Windows Runtime class, you get error 0x80131040: The located assembly’s manifest definition does not match the assembly reference.

What’s going on, and more importantly, how do I fix it?

Here’s what’s going on: Visual Studio checks the *Min version* of the C# project. This version controls which version of .NET Core and .NET Native are used. In mixed-language scenarios, such as we have here with C++/WinRT and C#, Visual Studio defaults to .NET Core 1.1 and .NET Native 1.4. However, if the minimum version is set to Windows 10 version 1709 (Build 16299) or higher, then Visual Studio copies the .NET Core 2 libraries into the application output folder and tries to run them against .NET Core 1.1.

That is the version mismatch that’s being reported. The mismatch is not with the version of your C# component. The mismatch is with the version of .NET Core.

The workaround is to set your C# component’s minimum version to Windows 10 version 1703 (Build 15063) or lower: From your C# project, go to Properties, Library, and under *Targeting*, set the *Min version* to *Windows 10 Creators Update (10.0; Build 15063)* or lower.

My colleague Johan Laanstra found another workaround, which has been shared by another colleague Alexander Sklar:

> 1. Right click on the VCXProj file → Manage NuGet Packages.
> 2. Search for **Microsoft.Net.Native.Compiler**, and install it.
> 3. Then add the following properties to the VCXProj

```
<PropertyGroup>
  <UseDotNetNativeToolchain
Condition="'$(Configuration)'=='Release'">true</UseDotNetNativeToolchain>
  <DotNetNativeVersion>2.2.3</DotNetNativeVersion>
</PropertyGroup>
```

And it looks like Carcadio Garcia found a different, lengthier workaround, which I haven't tried.

Bonus chatter: Tom McDonald informed me that Carcadio Garcia's workaround has not been updated for Visual Studio 16.6, which uses different directories from earlier versions of Visual Studio. Support for Visual Studio 16.6 can be found in this sample project. He also shared with me some cute little tables. Here's one of them:

Microsoft.NETCore.Universal-WindowsPlatform	Visual Studio	DotNetNative-Version	DotNetNative-SharedLibrary
6.2.10 (current)	16.6	2.2.8-rel-28605-00	2.2.27912.00
6.2.9	16.5	2.2.7-rel-27913-00	2.2.27912.00

There's also an issue with where the NuGet packages are kept. They could be kept in the user's private NuGet package store or in the Visual Studio global package store. The danger of using the user's private NuGet package store is that the user might not have the package. The danger of using the Visual Studio global package store is that the contents of that store are not contractual and can change at the next update.

NugetPath	Visual Studio	Sample project
$(ProgramFiles)\Microsoft SDKs\UWPNuGetPackages	≥ 16.6	Sample
$(USERPROFILE)\.nuget\packages	≤ 16.5	Sample

One trick is to create a blank C# UWP app targeting the matching version of the Microsoft.NETCore.UniversalWindowsPlatform meta-package and build it. That will restore all of the packages into the user's private package store so that other projects can consume them.

Raymond Chen

Follow