

# Dr.Bob's Prescriptions N°7: Delphi for .NET Preview, .NET assemblies and old (legacy) DLLs

42  
Bob-42.com

by Bob Swart

42  
Bob-42.com

Since my last column, Borland has announced and shipped Delphi 7 Studio. They have also included a preview version of the Delphi for .NET command-line compiler, without the VCL for .NET Preview (as semi-promised at BorCon in Anaheim), but at least with a little documentation and a number of newsgroups at forums.borland.com. (For more .NET information, be sure to check out the new special richplum.dotnet newsgroup on our news.dhms.net news server).

In this article, I want to example Delphi for .NET and looking a little bit at assemblies - the next generation of DLLs - and if and how we can take existing code with us to the .NET world.

## DLLs and assemblies

A dynamic link library can export routines, with arguments and optionally a result type. DLLs introduced the capability of writing parts of a system in different programming languages, and reusing code. Unfortunately, a DLL itself does not contain detailed information about the routines, their arguments and result types, so we had to use import libraries, header files or other ways to document the usage of the DLL. And when a DLL was accidentally replaced by another (newer or - worse - older) version, then all hell broke lose. Usually literally, because the user of the DLL no longer found the routines that were supposed to be there. The Component Object Model (COM) added new capabilities to this model, but was never able to solve it completely, perhaps because under the cover it was still built on the DLL mechanism.

A .NET assembly is the next logical step in the evolution of sharable and reusable building blocks. A .NET assembly does no longer contain mere routines, but it contains classes, with rich information - even more than the RTTI we've been used to in Delphi. In fact, .NET assemblies contain so much meta data, that it completely describes itself in detail, so we never have to use a header file, import unit or any other means to use a .NET assembly (bye bye HeadConv).

Apart from the fact that .NET assemblies contain meta data, we can also have more than one version of an assembly installed on our machine (usually in the Global Assembly Cache), and client applications can specify which one to use! No more DLL Hell where your application wouldn't run because a required MIDAS.DLL of Delphi 6.02 in the Windows\System32 directory was accidentally overwritten by the MIDAS.DLL of Delphi 6.01.

I'm sure that .NET assemblies are the way of the future. Having said that, I have years of work in the form of DLLs, and I do not want to move over to the new (.NET) world by abandoning everything from the old (Win32) world. Fortunately, we don't have to. It's not safe, and it's not managed code, but we can use unsafe, unmanaged Win32 DLLs in .NET applications, as I will show in this article. And while we're at it, I will also show an example of a .NET assembly, so enough introduction, let's get coding (and you need both a version of Delphi for Win32 and the Delphi for .NET Preview command-line compiler to play along).

## MyDLL

Let's start with a simple DLL called MyDLL, containing a single function TheAnswer, which the obvious return value of 42. Like the story behind the answer itself, it's not the question that counts, but the process.

```
library MyDLL;  
  
function TheAnswer: Integer; stdcall; export;  
begin  
    Result := 42  
end;  
  
exports  
    TheAnswer;  
  
begin  
end.
```

## UseDLL

Using this DLL can be done by importing the required routines from the DLL, which can be done with the "external" statement (resulting in an implicit link of the DLL, loading the DLL as soon as the executable starts), or with an explicit link, using LoadLibrary and GetProcAddress. The code here uses the external statement:

```
program UseDLL;  
{ $APPTYPE CONSOLE }  
  
function TheAnswer: Integer; external  
'MyDLL.dll';  
  
begin  
    writeln('The Answer is: ', TheAnswer)  
end.
```

## MyASM

An assembly in .NET with a similar functionality can be seen in the following listing. Note that .NET assemblies contain classes, and not just methods. The class is called LifeTheUniverseAndEverything (for those of you who may still not know where TheAnswer of 42 come from). Note that in the .NET world it's no longer common to start types with the "T" prefix, as Delphi developers have done for years. We should slowly try to learn to specify class types without the T prefix. Especially since our .NET assemblies will also be usable by other .NET development environments and languages like C#, and if you want to be a good .NET citizen, you have to play by the rules!

So it's class `LifeTheUniverseAndEverything` with a required constructor `Create` and method called `TheAnswer`.

```
library MyASM;
type
  LifeTheUniverseAndEverything = class
    constructor Create;
    function TheAnswer: Integer;
  end;

constructor
LifeTheUniverseAndEverything.Create;
begin
end;

function
LifeTheUniverseAndEverything.TheAnswer:
Integer;
begin
  Result := 42;
end;

end.
```

We can compile `MyASM.dpr` (containing this source code), using `dccil`, resulting in `MyASM.dll` - just like a "normal" DLL, with the exception that a normal DLL cannot export classes, but only routines.

## UseASM

Using the `MyASM` .NET assembly in a Delphi for .NET application is very simple. But look closely at the following source code, since there are more things going on behind the scenes than it may seem.

```
program UseASM;
{$APPTYPE CONSOLE}
uses
  MyASM;

var
  X: LifeTheUniverseAndEverything;
begin
  X := LifeTheUniverseAndEverything.Create;
  writeln('The Answer is: ', X.TheAnswer)
end.
```

Notice the `MyASM` "unit" in the `uses` clause? This seems to indicate that we want to use the unit `MyASM`, while there is no unit `MyASM` but only an (external) assembly called `MyASM`. The strong point about .NET assemblies is that they contain so much meta data, that they can be used just as if they were a unit in Delphi (or any other language on .NET, such as C#). The only "trick" that we have to perform next is to tell the `dccil` compiler to actually find and use that particular assembly - so `dccil` doesn't have to start looking for the `MyASM` unit, but for the `MyASM` assembly instead. This is done using the `-LU` flag of the compiler, so the `UseASM.dpr` project has to be compiled as follows:

```
dccil -LUMyASM UseASM.dpr
```

Unfortunately, this will give you an error message: "Fatal: Required package 'MyASM' not found", since apparently `dccil` cannot locate the `MyASM.dll`. This problem is actually due to the "preview" nature of the `dccil` command-line compiler, and something that will be solved in an upcoming update (or next preview or whatever). For now, the simple solution consists of - temporarily - copying the `MyASM.dll` file to the `C:\WinNT\Microsoft.NET\Framework\v1.0.3705` directory (or whichever directory also holds your `mscorlib.dll`). After that, the compile will succeed, and you can run the resulting `UseASM` executable to get the same result as before. But this time as safe, managed, .NET code using a .NET assembly, and all written in Delphi for .NET.

## UseDLLdotNet

Now, let's assume for a moment that we want to use some existing - legacy - code, only available as DLL (and we don't have the time to migrate all DLLs to .NET just yet, so we want to use them as-is today). In that case, you are probably pleased to learn that we can still import functions from DLLs like we did in the "old" days. The next code shows how to import a function from an external DLL called `MyDLL.dll`, which compiles and works without problems.

```
program UseDLLdotNet;
{$APPTYPE CONSOLE}
uses
  System.Runtime.InteropServices;

function TheAnswer: Integer; stdcall; external
'MyDLL.dll';

begin
  writeln('The Answer is: ', TheAnswer)
end.
```

Of course, you have to realise that the `MyDLL.dll` contains unsafe and unmanaged Win32 code, which could break the system.

Apart from importing a Win32 DLL the old way, we can also import it using a new functionality in the Delphi for .NET language: the `DllImport` attribute. Using this attribute, we can specify the DLL from where to import the following routine, as well as optional information like calling convention, character encoding, etc. The following code illustrates this:

```
program UseDLLdotNet;
{$APPTYPE CONSOLE}
uses
  System.Runtime.InteropServices;

[DllImport('MyDLL.dll')]
function TheAnswer: Integer; external;

begin
  writeln('The Answer is: ', TheAnswer)
end.
```

This is a convenient way to reuse existing code in your .NET applications. In fact, the Delphi for .NET Preview command-line compiler already uses this in the `Borland.Win32.Windows.pas` unit, which imports the existing Win32 API and makes it available in your .NET applications. As unsafe, unmanaged imports and calls, but available nevertheless. Which also means that it doesn't hurt to have some good Win32 API reference books at hand, such as the *Tomes of Delphi: Win32 Core API* and *Win32 Shell API*, which are reviewed in this magazine.

## Conclusion

In this article, I have not really focussed in-depth on the difference between assemblies and DLLs, but I did show how we can create both, and how we can use both in .NET applications (although using unsafe, unmanaged code in Win32 DLLs is of course only meant as a temporary solution, until you've rewritten your unsafe, unmanaged DLLs as safe, managed assemblies, replacing any non-supported language features or code constructs. However, it also means that choice between Win32 or .NET isn't really a one-or-the-other choice, it's more a migration from Win32 to .NET which doesn't have to be done overnight. If you haven't started any .NET specific work, then you should start now, and still be able to use most of your existing Win32 code as unsafe, unmanaged DLLs or COM objects.