# Getting Started with VB.Net using ExcelDNA

This is a collection of notes on learning ExcelDna.

A worked example is in the document “Practice project converting an Add-In to VB.NET with ExcelDna” using the CreateTOC.xlam example.

The latest Excel-DNA version is available on the CodePlex site: <http://exceldna.codeplex.com> also has links to tutorials

<http://groups.google.com/group/exceldna> Primary support channel

All applications use the ExcelDna.xll addin. The stages of learning successively add files:

1. Using only a text file MyAddin.dna which includes the source code text.
2. Add an external dll which you create from VB.net source code and compile using either the vbc.exe compile or the Visual Basic Studio IDE. I used the free Express edition.
3. Ease the transition using the official MS Primary Interop Assemblies (PIA) or third party libraries such as NetOffice
4. Simplify the deployment by packing the components into one .XLL file.

### Stage 1: using only a .DNA test file

Getting Started with ExcelDna - Do this first

<http://exceldna.codeplex.com/wikipage?title=Getting%20Started>

1. Create a user-defined function in Visual Basic

<http://www.blog.methodsinexcel.co.uk/2010/09/03/excel-dna-hello-world-function/>

Similar to above. Ideal if you want to write User Defined Functions (UDF), or want to copy a UDF from VBA to DNA for a quick and easy-ish performance improvement.

### Stage 2: Compiling a .DLL

<http://www.blog.methodsinexcel.co.uk/2010/09/22/writing-an-excel-dna-function-using-an-ide/>

Ross McLean uses the SharpDevelop IDE; VB Express is similar.

One gotcha that held me up a little was the need to have NO spaces around the equals sign in this line:

<DnaLibrary RuntimeVersion= "v4.0">

### Debugging

http://msdn.microsoft.com/en-us/library/wx0123s5.aspx

Visual Studio automatically chooses the Debug configuration when you choose Start from the Debug menu and the Release configurations when you use the Build menu.

You can tailor this from the menu Build > Configuration Manager.

VB Express does not show the option in the UI to debug an addin executing inside Excel but you can do it yourself by editing MyApp.vbproj.user file as follows:

<?xml version="1.0" encoding="utf-8"?>

<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

<PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">

<StartAction>Program</StartAction>

<StartProgram>C:\Program Files\Microsoft Office\Office14\EXCEL.EXE</StartProgram>

<StartArguments>MyApp.xll</StartArguments>

</PropertyGroup>

<PropertyGroup>

<ProjectView>ShowAllFiles</ProjectView>

</PropertyGroup>

</Project>

### Stage 3: Adding Intellisense to a DNA solution

<http://www.blog.methodsinexcel.co.uk/2010/10/28/ading-intellisense-to-a-dna-solution/>

Download the PIA .Zip file from the link in that post and extract to your hard disk. Setup up a DNA project in the normal way, go to the “Project” menu, “Add reference”, then “.Net assembly Browser” then go and find the PIA folder you just extracted, and pick the right version for your needs.

Also see NetOffice, a version-independent set of Office interop assemblies put together by Sebastian Lange.

<http://excel-dna.net/2012/01/30/excel-vba-to-vb-net-with-excel-dna-and-netoffice/>

<http://netoffice.codeplex.com/>

<http://groups.google.com/group/exceldna/browse_frm/thread/f356a30adf9130d0/c7625c8836affd3d>

### Stage 4: Simplify deployment by packing the components into one .XLL file.

ExcelDnaPack is a command-line utility to pack ExcelDna add-ins into a single .xll file.

Example: ExcelDnaPack.exe MyAddins\FirstAddin.dna

The packed add-in file will be created as MyAddins\FirstAddin.xll.

To pack additional assemblies, you add some entries like

<Reference Path="..." Pack="true" />

in the .dna file.

## Going further to C#

Charles Williams recommends this automatic code converter to ease the transition from VB to C# and C++ :

<http://tangiblesoftwaresolutions.com/>

## Background reading

Currently (Feb 2012) the only documentation on ExcelDna is effectively the google group.

Note that <http://exceldna.typepad.com> is old, not updated since 2006.

Documentation extracted from http://groups.google.com/group/exceldna

On Feb 2, 11:36 pm, Govert van Drimmelen <gov...@icon.co.za> wrote:

With Excel-DNA you can talk to Excel using either

1. using the C API or

2. using the COM object model.

If you're coming from VBA, the COM automation interface will be more familiar. So I'll focus on option 2.

Excel-DNA has no built-in definitions for the COM object model types like 'Range' and 'Worksheet'. However, Excel-DNA gives you a way to get to the 'root' Application object that you need for any other

access to the COM object model - just call ExcelDnaUtil.Application and you'll get an Application object that refers to the instance of Excel hosting your add-in. (Something like CreateObject("Excel.Application") may or may not give you the right Excel instance.)

From your VB.NET add-in you can now talk to the COM automation types either:

2(a) Late-bound. Your variables are types as 'Object' (the .NET equivalent of Variant), either explicitly or implicitly by not giving a type. Then code like this will work:

Dim xlApp

Dim myRange

xlApp = ExcelDnaUtil.Application

myRange = xlApp.Range("A1")

The disadvantage is that you have no intellisense and no checking at compile time.

2(b) Early-bound using an interop assembly. In this case you reference a .NET assembly that contains definitions for the COM object model. This interop assembly defines the types like 'Range' and 'Worksheet'

to your .NET code.

There are two options for the interop assembly:

2(b)(i) Use the official version-specific Primary Interop Assembly (PIA). This is where the namespace Microsoft.Office.Interop.Excel comes from. You can download and install the Office 2010 versions

here:http://www.microsoft.com/download/en/details.aspx?id=3508. Once installed, you'll be able to add references to the assembly called 'Microsoft.Office.Interop.Excel', where the 'Range' and 'Worksheet' types are defined.

2(b)(ii) Use a version-independent interop assembly like the NetOffice assemblies.

In both cases you need to make sure that you use the ExcelDnaUtil.Application object as the root of your access to the object model.

----------------

The type ExcelDna.Integration.ExcelReference is defined by Excel-DNA as a thin wrapper around the C API datatype used to indicate a sheet reference. In the sense that it denotes a part of a sheet, it is similar to the COM Range type, but to actually use the ExcelReference type you would typically pass it as an argument to some other C API calls. The only helper methods currently there are methods to to get and set data in the referenced area.

----------------

I hope I'm not making it all too complicated.

My own preference for the moment is to use and suggest the NetOffice assemblies, which is why I was exploring that path with the RiskGen porting effort.

On Feb 3, 11:50 am, Govert van Drimmelen <gov...@icon.co.za> wrote:

No larger document yet, but plenty of posts on the Google group if you search for any of the keywords.

------

If you use any of the interop assembly options (NetOffice or PIA) you can say:

Dim ws As Worksheet

Then you have to have some definition of that type at runtime too, so you need to have the interop assembly at the client too (however, see the 'Embed Interop Types' option below).

------

Otherwise, if you have no interop assembly referenced, you can say

Dim ws As Object

or equivalently (and also compatible with VBA, as you understand)

Dim ws

Not having an interop assembly means no intellisense and no checking for mistakes at compile time. So you could have a mistake like:

Dim myRange

myRange = Application.ActiveSheet.Ragne("A1")

and the error would only appear at runtime.

The performance of the late-binding (particularly from VB.NET) is very good, so not really an issue. And there is no deployment issue since you are not referencing additional libraries.

The PIA assemblies are installed in the Global Assembly Cache (GAC) so you should reference then from the .NET tab of the Add Reference dialog, and look for "Microsoft.Office.Interop.Excel" and "Office" -so do not browse to them explicitly. They should not be copied to your output directory either, since they live in the .NET GAC. To deploy to another machine, you need to run the downloaded installer for the Office Primary Interop Assemblies, which puts the PIA assemblies in the GAC and does some registration in the registry.

For the NetOffice assemblies you can just copy them somewhere, Add Reference and Browse there and Copy to Output. Then with the packing you can put them inside the .xll, so no other files or registration would be needed.

-----------

Actually if you are using .NET 4 (Visual Studio 2010) and targeting a single Excel version, say Excel 2010, there is another option I have not mentioned yet. The option was added in .NET 4 to embed some interop information in the assembly that you compile. I have not tried this myself, but it might work well for you. To do this you reference the PIA (via the .NET tab on Add References, not browsing) and then in

the Reference properties you set "Embed Interop Type: True". That should put the required information in your compiled assembly, and then you don't have to distribute the interop assemblies to other users.

This will only work under .NET 4, and probably won't work with the NetOffice assemblies since they are not 'Primary' interop assemblies.

<http://groups.google.com/group/exceldna/browse_frm/thread/7223e5ab0945e4/48d521ee114f667d?lnk=gst&q=vba#48d521ee114f667d>

RiskGen project example

1. I created the GlobalHelper class to see how compatible the VB.NET can be made, so that I have to change as little as possible from the VBA code. VB.NET makes the members of a Module available throughout the project, as VBA does to some of the classes in a COM type library. In Excel VBA the Application class is global (you can see its members in the <globals> part of the COM type library in VBA). Likewise all the enum members in the type library can be accessed as global constants.

So, as you point out, the GlobalHelper allows you to keep the VB.NET as ActiveSheet.XXX instead of Application.ActiveSheet.XXX. It's not something you need, but I wanted to experiment a bit to figure out what could make the VBA -> VB.NET transition easier.

For this example I just filled in the methods and enums in the GlobalHelper as I went along. I discussed a bit with Sebastian Lange (of NetOffice) how one could generate the GlobalHelper class automatically from the type library. It sounded promising.

2. & 6. The way COM properties are exposed from the NetOffice assemblies currently aren't great. This is because NetOffice is generated as C# code, and C# has some limitations in how properties can be defined. In particular, indexed properties and those that take parameters can't be defined in C# to match the way they can be defined in VB.NET or the COM interop libraries. This means some properties, like UsedRange, need to be called as get\_UsedRange in the current version of NetOffice. I'm hoping Sebastian can improve the property handling in a future version, maybe by generating VB.NET code for his interop assemblies. This issue wasn't really obvious before, since the way C# versions before C# 4 dealt with these properties was problematic, and needed the explicit get\_XXX and set\_XXX calls. VB.NET is flexible enough to define and consume these properties correctly, and C# 4 can consume them but not define them.

I think the reason the ActiveSheet.UsedRange is not flagged in the code snapshot is that I switched between different versions of NetOffice while putting the sample together. I tried to use a newer version with some improved property support, but had to go back to the one that only supports get\_UsedRange later.

3. The Visual Studio Express editions are not really full-featured development environments (though I think nicer than the VBA IDE). In particular they don't support Visual Studio add-ins, so I suspect your CodeRush won't work there. If you are looking for a free IDE with many features, have a look at SharpDevelop (be sure to get version 4.1 or later). It keeps getting better and I'm happy to support any issues you have with Excel-DNA in SharpDevelop.

7. If you are just targeting Excel 2010 you can certainly use the Primary Interop Assemblies (PIA) instead of the NetOffice assemblies. The Excel 2010 PIA assemblies are here: http://www.microsoft.com/download/en/details.aspx?id=3508. An advantage is that all the property stuff is already right in the PIAs. A disadvantage is that you might need to install the PIA assemblies using the downloaded installer - I'm not sure you can just copy or pack them into the .xll like the NetOffice assemblies. This also means you need admin rights to deploy your add-in. In general the PIA assemblies have a reputation for bringing deployment headaches (http://blogs.msdn.com/b/vsto/archive/2008/05/20/common-pitfalls- during-pia-deployment-and-installation.aspx).

It's easy enough to try though, apart from the get\_XXX and set\_XXX property issues (that NetOffice has and the PIA assemblies don't), you only need to change the namespace Imports.

If deployment is not a big concern, and you are targeting a single Excel version, the PIAs are probably the best choice at the moment. NetOffice might add some more nice features in future though, so it's worth keeping an eye on even if you don't use it.

## Microsoft and other sources

<http://blogs.msdn.com/b/pstubbs/archive/2004/01/15/59049.aspx>

Convert VBA to VB .Net and C#

<http://msdn.microsoft.com/en-us/vstudio/ms788236>

Free Book - Upgrading Microsoft Visual Basic 6.0 to Microsoft Visual Basic .NET

<http://www.upsizing.co.uk/Art52_VBAToNet2.aspx>

Converting Access VBA to VB.NET – General Principals

<http://msdn.microsoft.com/en-us/library/aa192490%28v=office.11%29.aspx>

Converting Code from VBA to Visual Basic .NET

Office 2003

For more information, see Introduction to Visual Basic .NET for Visual Basic Veterans.

<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vbcon/html/vboriintroductiontovisualbasic70forvisualbasicveterans.asp>

Many of the methods and properties of Office objects return the type Object, and you must explicitly convert the object to the correct type, as shown in the following example:

' Using Ctype to convert the Object.returned by Sheet1 to a Worksheet.

MsgBox(CType(ThisWorkbook.Worksheets("Sheet1"), Excel.Worksheet).Name)

Default Properties. In Visual Basic .NET, default properties are only supported if the properties take arguments. In VBA, you can use shortcuts when typing code by eliminating the default properties.

ByVal, ByRef Parameters. In VBA, parameters are passed by reference by default. In Visual Basic .NET, parameters are passed by value by default

Enumerations. Enumeration constants must be fully qualified in Visual Basic. NET.

In VBA, you can simply assign the constant because the enumeration is global to your project:

Selection.Find.Wrap = wdFindContinue

In Visual Basic .NET, you must fully qualify the constant with the enumeration name:

ThisApplication.Selection.Find.Wrap = Word.WdFindWrap.wdFindContinue

Type Excel.XL to scroll through the list of available enumerations for Excel

•Non-Zero Bound Arrays. In VBA, the default lower bound of an array dimension is 0 (zero). Using Option Base, you can change this to 1. In Visual Basic .NET, the Option Base statement is not supported, and the lower bound of every array dimension must be 0. Additionally, you cannot use ReDim as an array declaration. One thing to keep in mind when working with Office collections from Visual Basic .NET is that the lower array bounds of most Office collections begin with 1.

•Use of Parentheses with Method Calls. In VBA, parentheses are optional in some cases when you call subroutines, and it is sometimes difficult to remember when they are required. In Visual Basic .NET, parentheses are required when passing parameters in a method call.

•Set Keyword. In VBA, the Set keyword is necessary to distinguish between assignment of an object and assignment of the default property of the object. Since default properties are not supported in Visual Basic .NET, the Set keyword is not needed and is no longer supported.

•Data Access. Data binding to a Data Access Object (DAO) or Remote Data Object (RDO) data source is not supported in Microsoft Visual Basic .NET. ActiveX® Data Objects (ADO) data binding is supported for backward compatibility; however, you may want to consider converting to ADO.NET. For more information, see Comparison of ADO.NET and ADO.

Conversion of UserForms to Windows Forms. VBA UserForms cannot be copied or imported into Visual Studio .NET. In most cases, you will need to recreate your forms as Windows Forms. The use of drag-and-drop controls is consistent with creating forms in VBA, but event handlers for Windows Form controls are handled differently. Many new features are available to make creating your forms easier than before, for example:

•Control anchoring is now possible, so that when a user resizes your form, the controls automatically resize and reposition properly.

•Setting tab order is much easier with Windows Forms. Enable tab ordering by clicking Tab Order on the View menu. Then simply click each control in the preferred order.

•Creating menus in-line is an improvement over menu creation in VBA.

•In VBA, you can show a form as vbModal or vbModeless. In Visual Basic .NET, the ShowDialog method is used to display a form modally; the Show method is used to display a form non-modally. Note, however, that the form will display non-modally, but when you click in the Word or Excel document, the form moves to the background, which can be confusing to your users.

•Many new form controls are also available in Visual Basic .NET, such as data-entry validators, common dialog boxes, hyperlinked labels, system tray icons, panels, numeric-up/downs, on-the-fly designable tree views, Help file linkers, ToolTip extenders, and more.

http://msdn.microsoft.com/en-us/library/aa168292(v=office.11).aspx

Understanding the Excel Object Model from a .NET Developer's Perspective

<http://msdn.microsoft.com/en-us/library/kehz1dz1(vs.71).aspx>

Introduction to Visual Basic .NET for Visual Basic Veterans

<http://msdn.microsoft.com/en-us/library/kehz1dz1(v=vs.90).aspx>

Help for Visual Basic 6.0 Users

<http://oreilly.com/catalog/vbdotnetnut/chapter/appa.html>

Appendix A

What's New and Different in VB .NET

## Ribbon

This document shows both the VBA and the VB.NET (they call it Visual Basic) signatures for all the callbacks:

<http://msdn.microsoft.com/en-us/library/aa722523(v=office.12).aspx>

Customizing the 2007 Office Fluent Ribbon for Developers (Part 3 of 3)

<http://groups.google.com/group/exceldna/browse_frm/thread/60b7fac567c17505/b1c2612e25f74919?lnk=gst&q=vba#b1c2612e25f74919>

The onAction="RunTagMacro" will run a VBA macro (Sub) named in tag="MyVBAMacro" too (it does on my machine), but the onAction="MyMacro" attribute will only run methods on the Excel-derived class.

In your .dll you need to be sure that the class is ComVisible.

You can do this by marking the class as [ComVisible(true)], or by setting COM-Visible for the assembly:

Go to the project properties, and select the Application tab. Then click on Assembly Information and set the "Make assembly COM-Visible" checkbox. (This sets the assembly-wide [assembly:ComVisible(true)] attribute in your AssemblyInfo.cs file.)

By default (if there is no assembly-wide attribute) assemblies are ComVisible, which is why the code in the .dna file works. But the Visual Studio library template sets the assembly attribute to false by default. The assembly-wide attribute can always be overridden by an attribute on the particular class.

(Note that COM-Visible is a different setting to "Register for COM Interop" which you should never set for an Excel-DNA library.)

Serge says: I was having the same problem and it turns out that in Excel 2007 (may apply to other versions) if a Ribbon at some point causes Excel to crash, all custom Ribbons will be disabled in the future. To re-enable custom ribbons, I had to go to Excel Options => Add-Ins => Manage: Disabled Items Go... => Click on the "ribbonhelper" item and click Enable.