Taking the Step from VBA Macros to Autodesk® Inventor® Add-Ins

Brian Ekins – Autodesk

CP218-1 This session will introduce you to the world of Inventor add-ins. It begins with a look at Microsoft® Visual Basic® Express, a free programming interface that can be used to write add-ins. We'll also look at the differences between VBA and VB.Net. Next we'll discuss what an add-in is and its advantages and disadvantages compared to using VBA. Finally, we'll look at the process of converting a VBA macro into an add-in command.

About the Speaker:
Brian is a designer for the Autodesk Inventor programming interface. He began working in the CAD industry over 25 years ago in various positions, including CAD administrator, applications engineer, CAD API designer, and consultant. Brian was the original designer of the Inventor® API and has presented at conferences and taught classes throughout the world to thousands of users and programmers.

brian.ekins@autodesk.com

If you've done any programming with Inventor you've likely heard about Add-Ins. This paper is written for those users of Inventor that have some experience writing VBA macros and would like to have a better understanding of what an Add-In is and what's involved to create one. The following questions are addressed:

1. What is an Add-In?
2. Why would you want to create an Add-In?
3. How do you create an Add-In?
4. How do you convert your VBA macros?
5. How do you execute an Add-In command?
6. How do you debug an Add-In?
7. How do you deploy an Add-In?

What is an Add-In?

Technically an Add-In is a COM component. What this means is that it's a programming component that can be used by other programs. It exposes a programming interface to allow other programs to talk to it. In the case of an Inventor Add-In the other program that will talk to it is Inventor.

When an Add-In is installed it adds information into the registry identifying it as a COM component and also as an Inventor Add-In. When Inventor starts up, it looks in the registry to find the Add-Ins and then starts and interacts with each one. The interaction between Inventor and the Add-In at this point is minimal but consists of Inventor starting the Add-In and passing it the Inventor Application object. Through the Application object the Add-In has access to the full Inventor API and can perform whatever tasks the API allows.

During this startup process the Add-In typically hooks up to various Inventor events and then while Inventor is running, it just waits for an event to occur. For example, during startup it might create a button and connect to the button's OnExecute event. When the button is pressed by the end-user the OnExecute event is fired to the Add-In and then it can perform whatever action is associated with that button.

Why create an Add-In?

To better understand if an Add-In is appropriate for you let's look at some of the advantages and disadvantages when compared to VBA.

Advantages of an Add-In

1. **Loads at Inventor's Startup**

   This is one of the biggest advantages of an Add-In. It allows the Add-In to be running at the beginning of an Inventor session where it can connect to events to allow it to react to any action that occurs in Inventor. For example, a PDM system might want to know whenever a document is saved.

   Because an Add-In is loaded at start-up, it allows you to create functionality that is very tightly integrated with Inventor. From an end-user's perspective there's no difference between a native Inventor command and a well written Add-In command.

2. **Alternative to Document Automatic VBA Macros**

   Inventor supports the ability to create a VBA macro that's automatically run at certain times. There are code management and resource problems with using this functionality. For most cases where document auto macros are being used an Add-In is a better choice. (See the post titled **Converting VBA Auto Macros to an Add-In** at http://blogs.autodesk.com/modthemachine for more information.)

3. **Can Use New Languages**
   An Add-In can be created using any language that supports the creation of a COM component. This gives you the choice of choosing something more familiar than VBA if you have experience in other languages. This also gives you the opportunity to use the latest generation of programming tools. VBA and VB 6 are older technology and are being replaced by the newer .Net technologies. In a transition like this there are always pros and cons but I believe in this case the pros outweigh the cons because there are a lot of productivity enhancements in the .Net languages. For this paper I'll use Visual Basic 2008 Express Edition and the Visual Basic that's part of Visual Studio as the programming language.

4. **Better User-Interface**
   An Add-In can provide a much richer user-interface than what you can do with VBA. For example, the languages you use to create an Add-In have much better dialog creation tools. An Add-In can also control where its commands exist within Inventor's toolbars and can even create its own environments.

5. **Deployment**
   To provide the functionality of an Add-In to someone else, you just supply the installer and have them install it. The Add-In's buttons will automatically be available in the next session of Inventor.

   If you want to share a VBA macro you need to provide the source code to allow the user to insert it into their VBA project. If they want a button to run the macro they need to manually create it.

6. **Easier to Manage the Source Code and Executable**
   Source code of Add-Ins is easier to manage than VBA programs. VB.Net source code is saved as ASCII files which allows them to be easily searched. They can also be managed by source control systems. VBA programs are stored within .ivb files which are binary and need to be opened by Inventor to be read.

   Because the result of creating an Add-In is a dll you can version your programs. It's an easy matter for an end-user to check the version of the dll and know if they have the latest version of your Add-In. There isn't any versioning for VBA macros.

7. **Code Security**
   To share VBA macros you need to share your source code. It is possible to share an .ivb file and password protect the contents but this protection is not very secure. With an Add-In you're delivering a dll, not any source code.

8. **Better support for transactions and transcripting**
   A more advanced topic is *transcripting*. A transcript is a recording of the actions the end-user has performed. The transcript can be replayed to mimic those same actions. Transcripting is used by Inventor QA as an internal testing tool but is also available to others to use. The architecture to support transcripting also provides a richer set of functionality for transaction handling. Writing programs to support transcripting is reasonably complex and is most appropriate for large, complex applications. The transaction functionality supported by the TransactionManager object is sufficient for most applications and is much easier to implement.

Advantages of VBA

1. **Easier to write**
   Most macro type programs are easier to write using VBA than VB.Net. VBA is simpler and was specifically designed for programming the type of programming interface that Inventor has.

2. **Better for rapid prototyping**
   For quickly testing or verifying the ability of the API to perform some function in Inventor, VBA is easier and faster. It's easy to write code, test it, make changes, and test it again.

   I'll frequently write small prototypes of an application using VBA and then convert it to VB.Net once I've been able to determine that what I want to do is feasible.
3. **Better Object Browser**
   The Object Browser in VBA provides a much cleaner and easier to understand view of the Inventor objects. I often use the VBA Object Browser even when I'm programming in VB.Net.
4. **Debugger is better at looking at Inventor objects**
   The VBA debugger provides better information when looking at Inventor objects. This is also a reason why VBA is better for rapid prototyping.

Should you abandon VBA and switch to writing Add-Ins? Possibly not. An Add-In is not the ideal solution to everything and each case needs to be looked at individually. If VBA is working for you now there's probably no compelling reason to make a big switch. If you tend to write smaller, simple utilities that only you or a small group use, VBA is still likely the best solution. If you've had issues with VBA because of its limitations, Add-Ins may be the solution you need.

How do you create an Add-In?

Before looking at the mechanics of creating an Add-In, here's what you need to create an Add-In.

What you'll Need

A Programming Language - To write an add-in you need to use a language that supports creating COM components. For this paper I'll discuss using either Visual Basic 2008 Express Edition or the Visual Basic that is part of Visual Studio 2008. Visual Basic 2008 Express Edition is particularly interesting because it is a free version of Visual Basic and can be downloaded from Microsoft's website. It's a great way for a new programmer to get started without making a financial investment. Microsoft also provides a lot of training material that is free with much of it targeted at the new programmer. If you're new to programming or the .Net languages it would be good to work through some of the Microsoft programming tutorials to familiarize yourself with the language and development environment.

The express edition is free but when you get something for free you expect some limitations. For Add-In development, the Express edition gives you everything you're likely to need with the exception of two things, which I'll show you how to work around later. You can always upgrade in the future to Visual Studio Standard Edition for $299 if you decide you need more functionality. Search Microsoft's site for the "Visual Studio 2008 Product Comparison Guide" for a complete comparison of the different versions.

Inventor SDK Developer Tools – You'll also need some Inventor specific development tools. These are delivered as part of a standard Inventor installation but you need to perform a couple of additional steps to make them available. Here are the steps, which are different depending on whether you're using Inventor 2009 or Inventor 2010.

Inventor 2010

1. You need to run the SDK installer to make the SDK files available. The location of the installer is different for Windows XP and Windows Vista or Windows 7.

   Windows XP: C:\Program Files\Autodesk\Inventor 2010\SDK\DeveloperTools.msi

   Windows Vista or 7: C:\Users\Public\Documents\Autodesk\Inventor 2010\SDK\DeveloperTools.msi

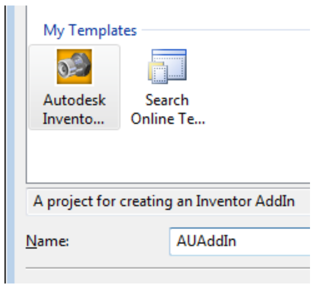
2. Next you need to install the Add-In wizard, which you do by running the InventorWizards.msi installer located in the DeveloperTools\Tools\Wizards directory that was created when you installed the developer tools.

Inventor 2009
If you're still using Inventor 2009 you'll need to manually install the template files. At the time Inventor 2009 was released Visual Studio 2005 was the current release, so installing the wizard as instructed above does not install it for Visual Basic Express Edition 2008 or Visual Studio 2008. You can easily work around this issue using the steps below.

1. Get the template file, *VBInventorAddInTemplate.zip*. You can get this file from the packaged set of files that go with this AU session or you can get it from my blog in the post titled "Converting VBA Auto Macros to an Add-In". (http://blogs.autodesk.com/modthemachine)
2. Copy the *VBInventorAddInTemplate.zip* file to the following location, (don't unzip it but just copy the zip file as-is): My Documents\Visual Studio 2008\Templates\ProjectTemplates

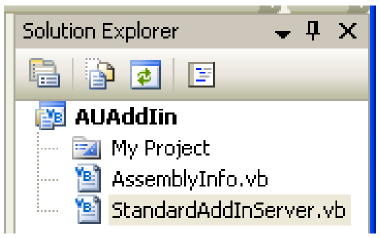
Creating an Add-In

You create an Add-In by starting VB 2008 Express or Visual Studio (both of which I'll just refer to as VB.Net for the rest of this paper, unless there's a difference between the two). Within VB.Net create a new project using the **New Project** command in the File menu. The New Project dialog displays the templates that are available to create various types of projects. Select the "Autodesk Inventor AddIn" template as shown to the right. If you don't see the icon for the Inventor Add-In template, make sure the top-level "Visual Basic" node is selected in the Project Type tree on the left of the dialog. Specify a name that makes sense for your project, (I'm using "AUAddIn" for this example), and click OK.



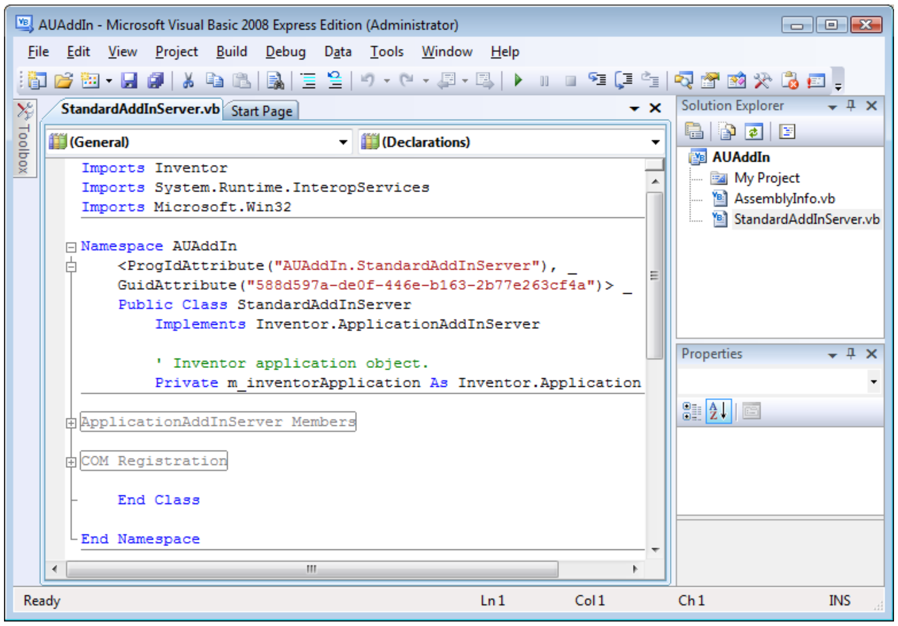
A project for creating an Inventor AddIn
Name: AUAddIn

Congratulations, you've just created an Add-In. Click the **Build *AUAddIn*** command (or the name or your Add-In) in the Build menu. Once it's finished building, start Inventor and look in the Add-In Manager where you should see your Add-In in the list. It's not very exciting at this point because the Add-In doesn't do anything, but the foundation is now in place to build on.



Let's look at what has been created for you. In the Solution Explorer you'll see that two files were created for the project; AssemblyInfo.vb and StandardAddInServer.vb. AssemblyInfo.vb contains information about the project and you won't need to do anything with this file. The interesting file is StandardAddInServer.vb.

If we look at the code for StandardAddInServer.vb you'll see what's shown below.



```
Imports Inventor
Imports System.Runtime.InteropServices
Imports Microsoft.Win32

Namespace AUAddIn
    <ProgIdAttribute("AUAddIn.StandardAddInServer"), _
    GuidAttribute("588d597a-de0f-446e-b163-2b77e263cf4a")> _
    Public Class StandardAddInServer
        Implements Inventor.ApplicationAddInServer

        ' Inventor application object.
        Private m_inventorApplication As Inventor.Application

ApplicationAddInServer Members

COM Registration

    End Class

End Namespace
```

At first glance it looks like there's not very much code, but most of it is hidden within two collapsed regions: "ApplicationAddInServer Members" and "COM Registration". Before we look at the code in those regions let's look at the code that is visible.

First, it imports three libraries, one of them being the Inventor library. (A reference to this library was also added to the project.)

Next it defines the class "StandardAddInServer" and specifies the Add-In's Class ID. The Class ID is the big number in the GuidAttribute line; "588d597a-de0f-446e-b163-2b77e263cf4a" in my example. This is the unique identifier for your Add-In. You'll see later how it is used when creating things that are associated with your Add-In.

The other important item is the declaration of the variable `m_inventorApplication`. It's only when you program with Inventor's VBA that you have access to the ThisApplication property. An Add-In doesn't have the ThisApplication property and needs to get access to the Application object in some other way. The m_inventorApplication variable is used to save a reference to the Inventor Application object.

Within the "COM Registration" region is the code that does the registration to identify this as an Inventor Add-In. Most of the code in this region should be left as-is but there are a couple of things you may want to change. The first instance of **`clsid.SetValue(Nothing, "AUAddIin")`** defines the name of your Add-In as it's shown in the Add-In Manager. You can change the "**`AUAddIin`**" portion to anything you want. The second instance of the statement **`clsid.SetValue(Nothing, "AUAddIin")`** is the description of your Add-In. The description is displayed at the bottom of the Add-In Manager. Both of these are highlighted in the code below.

The other section of SetValue methods that are highlighted below define which version of Inventor your Add-In will be loaded for. Notice that all but one of these lines is commented out. The commented lines illustrate the various options that are available. In this example, the Add-In will load for all versions of Inventor later than version 13 (Inventor 2009), so it will load for Inventor 2010 and beyond. If you're writing your Add-in to work with Inventor 2009 you should change the value to "12..". That finishes the registration code.

```
<ComRegisterFunctionAttribute()> _
Public Shared Sub Register(ByVal t As Type)

    Dim clssRoot As RegistryKey = Registry.ClassesRoot
    Dim clsid As RegistryKey = Nothing
    Dim subKey As RegistryKey = Nothing

    Try
        clsid = clssRoot.CreateSubKey("CLSID\" + AddInGuid(t))
        clsid.SetValue(Nothing, "AUAddIin")
        subKey = clsid.CreateSubKey("Implemented Categories\{39AD2B5C-7A29-
        subKey.Close()

        subKey = clsid.CreateSubKey("Settings")
        subKey.SetValue("AddInType", "Standard")
        subKey.SetValue("LoadOnStartUp", "1")

        'subKey.SetValue("SupportedSoftwareVersionLessThan", "")
        subKey.SetValue("SupportedSoftwareVersionGreaterThan", "13..")
        'subKey.SetValue("SupportedSoftwareVersionEqualTo", "")
        'subKey.SetValue("SupportedSoftwareVersionNotEqualTo", "")
        'subKey.SetValue("Hidden", "0")
        'subKey.SetValue("UserUnloadable", "1")
        subKey.SetValue("Version", 0)
        subKey.Close()

        subKey = clsid.CreateSubKey("Description")
        subKey.SetValue(Nothing, "AUAddIin")

    Catch ex As Exception
        System.Diagnostics.Trace.Assert(False)
    Finally
        If Not subKey Is Nothing Then subKey.Close()
        If Not clsid Is Nothing Then clsid.Close()
        If Not clssRoot Is Nothing Then clssRoot.Close()
    End Try
End Sub
```

The “ApplicationAddInServer Members” region is where you find the heart of the Add-In. This section of code is shown below. (I’ve removed most of the comments and simplified the code to save space.)

```
Public Sub Activate( ByVal addInSiteObject As ApplicationAddInSite, _
                     ByVal firstTime As Boolean)

    ' Initialize AddIn members.
    m_inventorApplication = addInSiteObject.Application
End Sub
----------------------------------------------------------------------------
Public Sub Deactivate()
    ' Release objects.
    Marshal.ReleaseComObject(m_inventorApplication)
    m_inventorApplication = Nothing

    System.GC.WaitForPendingFinalizers()
    System.GC.Collect()
End Sub
----------------------------------------------------------------------------
Public ReadOnly Property Automation()
    Get
        Return Nothing
    End Get
End Property
----------------------------------------------------------------------------
Public Sub ExecuteCommand(ByVal commandID As Integer)
End Sub
```

The four methods shown above (Activate, Deactivate, Automation, and ExecuteCommand) are required to be supported by all Add-Ins. The ExecuteCommand method is now obsolete so you’ll never use it. The Automation method is rarely used and is not discussed here. You will need to use the Activate and Deactivate methods.

The Activate method is called by Inventor when it starts the Add-In. This method has two arguments. The first argument passes in an ApplicationAddInSite object. The ApplicationAddInSite object is a very simple object that just supports the Application property. The Add-In uses to get the Inventor Application object and assigns it to the `m_inventorApplication` member variable so it’s available to the rest of the Add-In. The second argument indicates if this is the first time the Add-In has ever been run. This is used when creating the user-interface for your Add-In, as we’ll see later.

The Deactivate method is called by Inventor when the Add-In is being shut down. This gives the Add-In a chance to clean up and release the references it has to Inventor objects. An Add-In is shut down when Inventor is shut down or when the end-user unloads it through the Add-In Manager. We’ll look at what you need to do in the Deactivate method later.

How do you convert your VBA Macros?

When converting VBA code into an Add-In there are two things to consider. First, where does your VBA code go within the Add-In and second, what has to be changed to make the code work with VB.Net. Even though VBA and VB.Net are both variations of Visual Basic there are significant differences that you’ll need to be aware of.

Converting VBA Code

Converting VBA code to VB.Net is a copy and paste process. You set up the basic architecture of your program in VB.Net, create any needed dialogs, and then begin copying and pasting code from your VBA project into the VB.Net project. There are differences between VBA and VB.Net so some code will require editing to be valid in VB.Net. Some of these differences are easy to see and are pointed out as errors by VB.Net; others are not so easily found and show up either as run-time errors or incorrect results. It's best to copy and paste one function at a time so you can check the code and try to catch any of these potential problems. The issues I think you're most likely to encounter are listed below.

1. **The global variable ThisApplication isn't available in VB.Net.** It was shown earlier how an Add-In is able to get the Application object through the Activate method. You'll need to somehow make this available to the code you copied from VBA. There are two basic approaches; use a global variable or pass it in as an argument. Which approach you choose, is up to you.

   For this paper I've chosen to pass the Application object as an argument. I think this makes the code clearer and requires less editing of the VBA code. Below is an example of a VBA function before and after modifying it to handle this input argument. Notice that I used "ThisApplication" as the name of the argument so I don't need to edit the name of the variable within the function.

```
' Before
Public Sub Sample()
    MsgBox "There are " & ThisApplication.Documents.Count & " open."
End Sub

' After
Public Sub Sample(ThisApplication As Inventor.Application)
    MsgBox("There are " & ThisApplication.Documents.Count & " open.")
End Sub
```

2. **VB.Net requires fully qualified enumeration constants.** This means the statement below, which is valid in VBA, does not work in VB.Net.

```
oExtrude.Operation = kJoinOperation
```

   In VB.Net you must fully qualify the use of kJoinOperation by specifying the enumeration name as shown below.

```
oExtrude.Operation = PartFeatureOperationEnum.kJoinOperation
```

   These are easy to catch and fix since VB.Net identifies them as errors and IntelliSense does most of the work for you by creating the fully qualified name.

3. **Method arguments default to ByVal.** In VBA, arguments default to ByRef. Here's an example to illustrate what this means. The VBA Sub below takes a feature as input and returns some information about the feature.

```
Sub GetFeatureInfo( Feature As PartFeature, Suppressed As Boolean, _
                        DimensionCount As Long)
```

   In VBA this code works fine since it's optional to specify whether an argument is ByRef or ByVal and if you don't specify one it defaults to ByRef. A ByRef argument can be modified within the Sub and the modified value will be passed back to the calling routine. A ByVal argument can also be modified, but the value is local and is not passed back to the calling routine.

In this example the Suppressed and DimensionCount arguments need to be ByRef since they're used to return information to the caller. The Feature argument can be declared as ByVal since it's not expected to change. VB.Net requires you to declare each variable as ByRef or ByVal. If it isn't specified for an argument, VB.Net automatically sets it to ByVal when you paste in your VBA code. Because of that, this example won't run correctly because the Suppressed and DimensionCount arguments won't return the correct values. They need to be changed to ByRef arguments for the sub to function as expected.

4. **Arrays have a lower bound of 0 (zero).** I think this is the change that will result in the most work and errors when porting code from VBA to VB.Net. In VBA the default lower bound of an array is 0 but it's common to use the Option Base statement to change this to 1. It's also common in VBA to specify the lower and upper bounds in the array declaration as shown below.

```
Dim adCoords(1 To 9) As Double
```

In VB.Net the above statement is not valid. The lower bound of an array is always 0. The equivalent statement in VB.Net is:

```
Dim adCoords(8) As Double
```

This creates an array that has an upper bound of 8. Since the lower bound is zero the array can contain 9 values. This can be confusing for anyone familiar with other languages where the declaration is the size of the array rather than the upper bound. If your VBA program was written assuming a lower bound of 1, adjusting the lower bound to 0 shifts all of the values in the array down by one index. You'll need to change the index values everywhere the array is used to account for this.

5. **Arrays of variable size are handled differently in VB.Net.** Here are a couple of issues that you might run into. First, you can't specify the type when you re-dimension an array. Specifying a type will result in an error in VB.Net

```
' VBA
ReDim adCoords(18) As Double
```

```
' VB.Net
Redim adCoords(18)
```

Second, is that declaring an array in VB.Net does not initialize it. The VBA code below will fail in .Net with a type mismatch error. This is easily fixed by initializing the value to an empty array, as shown (open and closed braces).

```
' VBA
Dim adStartPoint() As Double
Dim adEndPoint() As Double
Call oEdge.Evaluator.GetEndPoints(adStartPoint, adEndPoint)
```

```
' VB.Net
Dim adStartPoint() As Double = {}
Dim adEndPoint() As Double = {}
oEdge.Evaluator.GetEndPoints(adStartPoint, adEndPoint)
```

6. **Some data types are different.** There are two changes here that can cause problems. First, the VBA type **Long** is equivalent to the VB.Net **Integer** type. If you're calling a method that was expecting a Long or an array of Longs in VBA, that same code will give you a type mismatch error in VB.Net. Change the declaration from Long to Integer and it should fix it.

   Second, the *Variant* data type isn't supported in VB.Net. If you have programs that use the Variant type, just change those declarations to the new *Object* type instead.

7. **Variable scope.** The scope of variables within functions is different with VB.Net. Variable scope is now limited to be within code blocks where-as VBA was only limited to within a function. If you copy the VBA function below, (which works fine in VBA), into a VB.Net program it will fail to compile. The last two uses, (underlined in the sample below), of the variable strSuppressed report that the variable is not declared. In this example strSuppressed is declared within the If Else block and is only available within that block.

```
' VBA
Public Sub ShowState(ByVal Feature As PartFeature)
    If Feature.Suppressed Then
       Dim strSuppressed As String
       strSuppressed = "Suppressed"
    Else
       strSuppressed = "Not Suppressed"
    End If

    MsgBox "The feature is " & strSuppressed
End Sub
```

   Here's a version of the same function modified to work correctly in VB.Net. The declaration of the strSuppressed variable has been moved outside the If Else block, and has scope within the entire sub.

```
' VB.Net
Public Sub ShowState(ByVal Feature As PartFeature)
    Dim strSuppressed As String
    If Feature.Suppressed Then
       strSuppressed = "Suppressed"
    Else
       strSuppressed = "Not Suppressed"
    End If

    MsgBox "The feature is " & strSuppressed
End Sub
```

8. **Events.** The concepts and basic mechanics of how to use events is the same in VB.Net but there are some enhancements to events in VB.Net. The change that will impact the conversion of your VBA code to VB.Net is a change in the signature of the event handler Sub. Here's an example of a VBA event handler is shown below.

```
' VBA
Private Sub oBrowser_OnActivate()
    ' Handling Code
End Sub
```

The handler for the same event in VB.Net is shown below. Notice the “Handles” keyword which is now used to specify that the Sub handles a specific event. In VBA the name of the Sub designated that it was an event handler. In VB.Net the name of the sub isn’t important.

```
' VB.Net
Private Sub oBrowserPane_OnActivate() Handles oBrowserPane.OnActivate
    ' Handling Code
End Sub
```

Because of this, rather than copy and paste the entire event handler sub from VBA it’s best to create a new empty sub in VB.Net and then copy and paste the contents of the sub from VBA.

9. **Other.** There are a couple of other changes you don’t need to do anything about but that you should be aware of. The Set keyword is no longer supported. It’s automatically removed from any lines when you paste it into VB.Net. This simplifies writing programs because in VBA it wasn’t always clear when you needed to use Set and when you didn’t.

   Parentheses are now required around property and method arguments. This was also a bit confusing in VBA because of their inconsistent use. When you copy and paste code into VB.Net it will automatically add any required parentheses. The Call statement is still supported but no longer needed.

Cool stuff in VB.Net

As we see from the above discussion, there are differences between VBA and VB.Net which cause some issues when porting code between them, however the fact that VB.Net is different is also good because it provides a lot of new capabilities that we didn’t have in VBA. Here’s a short list of some of my favorites. You can look them up in the VB.Net documentation for a complete description.

1. You can set the value of a variable when you declare it.

```
Dim partDoc As PartDocument = invApp.ActiveDocument
```

2. Error handling is much better in VB.Net. The old “On Error” type of error handling is still supported but you can now use the much more powerful Try Catch style of error handling.

3. All of the registration information is part of the dll you create. No more .reg files are needed like they were when an Add-In was created using VB oreinvl6.

4. Debugging an Add-In is easier than it was in VB 6.

5. The .Net libraries provide a much richer set of functionality for math functions, string handling, file handling, working with XML files, and most everything else. You’re now using the same libraries as anyone else that’s coding with any of the .Net languages.

6. Incrementing a variable. There’s now some simpler syntax for incrementing the value of a variable.

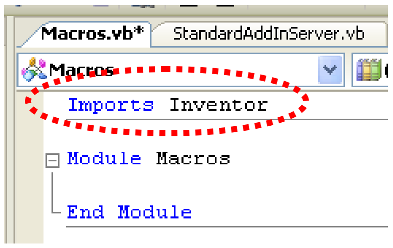
```
' VB 6
i = i + 1
```

```
' VB.Net
i += 1
```

Where to put your Code

I said earlier that to move code from VBA to VB.net you'll need to copy and paste it. The question now is where do you paste it? There isn't a single answer to this and there are a lot of right answers. Rather than discuss the various options I'll just choose one method that is easy and is somewhat similar to how code was organized in VBA.

Create a new code module in your VB.Net project using the **Add Module** command in the Project menu. In this example I named the module "Macros". Here's the code window after creating the module. Notice that the Imports line for the Inventor library has been added.



```
Imports Inventor

Module Macros

End Module
```

The VBA macros will be copied into this module. Below is an example of this module after a simple VBA macro has been copied in. Notice that I've added an argument to the macro to allow the Application object to be passed in.

```
Module Macros

  Public Sub FeatureCount( ThisApplication As Inventor.Application )
    Dim oPartDoc As PartDocument
    oPartDoc = ThisApplication.ActiveDocument

    MsgBox("There are " & oPartDoc.ComponentDefinition.Features.Count & _
           " features in this part.")
  End Sub

End Module
```

Converting VBA Dialogs

There isn't any support for converting VBA dialogs to VB.Net. You'll need to recreate them from scratch in VB.Net. However, this doesn't mean you can't reuse the code behind the dialog. For example, the code you wrote to react to a button click can be reused, but the physical button on the dialog will have to be recreated. Recreating your dialogs isn't necessarily a bad thing since VB.Net has better dialog tools and supports a much richer set of controls than were available in VBA.

You create a dialog as a Visual Basic form and typically display it in response to the user pressing a button. The code below illustrates responding to a button's OnExecute event by displaying a dialog. This example uses the Show method to display the form in a modeless state. You can also use the ShowDialog method which will display it as a modal dialog.

```
 Private Sub m_featureCountButtonDef_OnExecute( ... )
    ' Display the dialog.
    Dim myForm As New InsertBoltForm
    myForm.Show(New WindowWrapper(m_inventorApplication.MainFrameHWND))
End Sub
```

One issue with displaying a dialog is that by default it is independent of the Inventor main window. This can cause a few problems; the Inventor window can cover the dialog, when the end-user minimizes the Inventor window your dialog is still displayed, and key presses that represent keyboard shortcuts are stolen by Inventor. To get around these problems you can make the dialog a child of the Inventor window. The sample above does this by using the WindowWrapper utility class, shown below. Just copy the code below into your project and then you can use the WindowWrapper function as it's used above.

```
#Region "hWnd Wrapper Class"
' This class is used to wrap a Win32 hWnd as a .Net IWind32Window class.
' This is used for parenting a dialog to the Inventor window.
'
' For example:
' myForm.Show(New WindowWrapper(m_inventorApplication.MainFrameHWND))
'
Public Class WindowWrapper
    Implements System.Windows.Forms.IWin32Window
    Public Sub New(ByVal handle As IntPtr)
        _hwnd = handle
    End Sub

    Public ReadOnly Property Handle() As IntPtr _
      Implements System.Windows.Forms.IWin32Window.Handle
        Get
            Return _hwnd
        End Get
    End Property

    Private _hwnd As IntPtr
End Class
#End Region
```

How do you execute your Add-In macro?

VBA is designed for simple creation and execution of macros. You can select and run macros from the **Macros** command or you can create a button for a macro using the **Customize** command. An Add-In doesn't provide such an easy interface but does provide additional options and flexibility that you don't have with VBA. With the introduction of the ribbon interface in Inventor 2010 it becomes even more flexible but also more complex because of the additional options. For now, I'll keep it simple by looking at the minimum work needed to create a button that will execute the sub in your Add-In. The code below is for Inventor's classic user-interface. Because Inventor automatically converts any classic interface into the new ribbon interface this allows your Add-In to work in both interfaces. There is another paper that describes the details of the ribbon interface.

The first step is to create a ButtonDefinition object to represent your command. The ButtonDefinition object defines what your button will look like, (name, tool tip, description, icon, enabled state, etc.) and it supports the OnExecute event. The OnExecute event is fired whenever the end-user clicks the button. You create your ButtonDefinition objects in the Activate method of the Add-In, as shown below.

The ButtonDefinition object defines how a button looks and behaves but it's not the physical button that the end-user clicks. The clickable button is a CommandBarControl object. The CommandBarControl defines a position within the user-interface and is associated with a ButtonDefinition object that defines what the button looks like. You create your ButtonDefinition objects every time the Add-In is started. However, you only create the CommandBarControl objects the very first time the Add-In is run (when the firstTime argument is True). They're only created the first time because Inventor remembers their existence and position after that.

Below is some sample code that demonstrates all of this. I've also added a declaration and some code to get the client ID of the add-in and save it in a variable. This variable is used wherever a client ID is used. This helps to eliminate problems with different client ID's being using.

```
' Declare member variables.
Private m_inventorApplication As Inventor.Application
Private m_ClientID As String
Private WithEvents m_featureCountButtonDef As ButtonDefinition

Public Sub Activate(ByVal addInSiteObject As Inventor.ApplicationAddInSite, _
                    ByVal firstTime As Boolean)
    ' Initialize AddIn members.
    m_inventorApplication = addInSiteObject.Application

    ' Get the ClassID for this add-in and save it in a
    ' member variable to use wherever a ClientID is needed.
    m_ClientID = AddInGuid(GetType(StandardAddInServer))

    ' Create the button definition.
    Dim controlDefs As ControlDefinitions
    controlDefs = m_inventorApplication.CommandManager.ControlDefinitions

    m_featureCountButtonDef = controlDefs.AddButtonDefinition( _
            "Count Features", _
            "AUAddInCountFeatures", _
            CommandTypesEnum.kQueryOnlyCmdType, _
            m_ClientID, _
            "Count the features in the active part.", _
            "Count Features")

    If firstTime Then
        ' Create a new command bar (toolbar) and make it visible.
        Dim commandBars As CommandBars
        commandBars = m_inventorApplication.UserInterfaceManager.CommandBars
        Dim commandBar as CommandBar
        commandBar = commandBars.Add( "My Macros", "AUAddInMyMacros",,m_ClientID)
        commandBar.Visible = True

        ' Add the control to the command bar.
        commandBar.Controls.AddButton(m_featureCountButtonDef)
    End If
End Sub
```

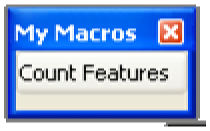
Here's a brief discussion of the arguments for the AddButtonDefinition as used above.

1. The first argument is the display name. This is used for the text on the button.
2. The second argument is the internal name. This is a unique identifier you define for this particular control definition. It must be unique with respect to all other control definitions in Inventor.
3. The third argument categorizes this command. The two most common categories are kQueryOnlyCmdType and kShapeEditCmdType. The first is for a command that just performs queries and the second is for a command that modifies geometry. There are others that are described in Inventor's programming help.
4. The fourth argument is the client ID of your Add-In. Note that it's enclosed within braces and reuses the class ID or your Add-In.
5. The fifth argument is the description. This appears within Inventor's status field.
6. The sixth argument is the tool tip.
7. There are some additional optional arguments that are not defined in this example that allow you to define an icon. Without specifying icons, the name of the command is displayed on the button. We'll look at icons later.

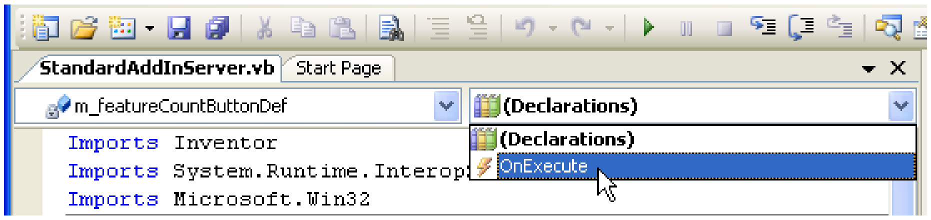
If you shut down Inventor, build your Add-In, and then restart Inventor you won't see any difference because this isn't the first time your Add-In has been run so the tool bar creation code isn't executed.

You can change one of the registry values of your Add-In to indicate it has a new user-interface and that Inventor needs to treat the next time it starts as its “first time”. Here’s a section of the Add-In registration that was discussed earlier. Edit the value of the “Version” value. It doesn’t matter what the value is as long as it’s different than the previous value.

```
        'subKey.SetValue("Hidden", "0")
        'subKey.SetValue("UserUnloadable", "1")
        subKey.SetValue("Version", 1)
        subKey.Close()
```



Now, if you run Inventor again you should see a new toolbar with a single button representing the new command. If you click the button, nothing happens. This is because you’re not listening for and responding to the button’s OnExecute event. To handle this event you need to set up an event handler. In the code window select the name of the object from the left-hand pull-down (m_featureCountButtonDef in this example) and then select the event you want to handle from the right-hand pull-down (OnExecute), as shown below.



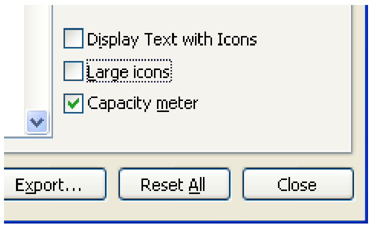
```
Imports Inventor
Imports System.Runtime.Interop
Imports Microsoft.Win32
```

The OnExecute event handler code is inserted into your project. The code below illustrates the code you write to call the FeatureCount sub whenever the button is clicked. The Inventor Application object is passed as the single argument.

```
Private Sub m_featureCountButtonDef_OnExecute( ... )
    FeatureCount(m_inventorApplication)
End Sub
```

Creating Icons for your Commands

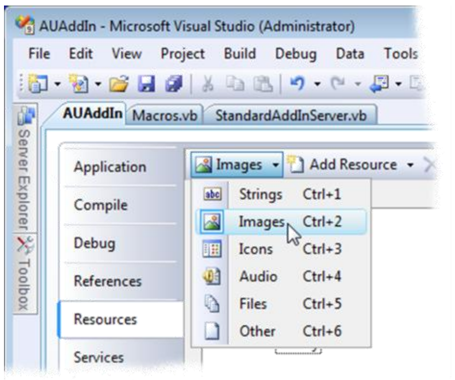
It’s also good to have an icon in addition to the command name to minimize the size of the button and make the look of your command consistent with the rest of the Inventor commands. Here are the steps for creating and using icons for your buttons. Visual Basic 2008 Express does not provide a built-in tool to create icons but you can use any graphics editor you want to create a .bmp or .ico file. There is a document delivered as part of the SDK that discusses the guidelines you should follow when creating icons. The file is: *SDK\Docs\Guidelines\ Design Guidelines (Icons).doc*



☐ Display Text with Icons
☐ Large icons
☑ Capacity meter

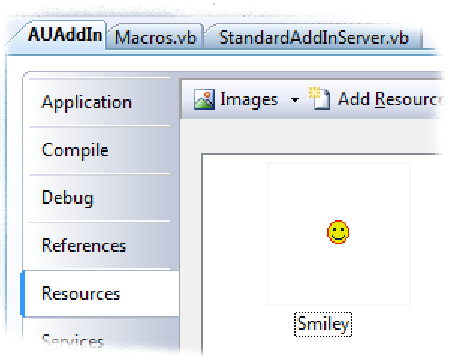
For Inventor’s classic user-interface, there are two standard sizes for icons, 16x16 and 24x24. The end-user can choose from the Toolbars tab of the Customize dialog whether they want large icons or not, as shown to the right. You can choose to only supply a small icon and Inventor will scale it to create a large one when needed, but the result is not as good as when you create the large icon yourself.

Icon design is somewhat of an art and can consume a lot of time to create something that looks good and represents the command. I’ve had my best luck designing icons by finding an existing icon, from Inventor or another application, which has elements similar to the command I’m creating and modifying it. I copy the icon by doing a screen capture and edit it to get the desired result.



For this example I used the Windows Paint program to create a 16x16 icon and saved it as a .bmp file. A .bmp file can be imported into a VB.Net project as a resource. To do this open the Properties page for your project by running the Properties command, (the last command in the Project menu). On the Properties page select the Resources tab on the left and then choose the type of resource from the pull-down. The picture to the right illustrates selecting the "Images" type. If you created an icon (.ico file) then you would choose "Icons".

Next, you can add the resource using the "Add Resource" pull-down and selecting the "Add Existing File…" option. Browse to and select your existing .bmp or .ico file to add it to the project. Next, assign a logical name to the resource. You can see below that I've named my image "Smiley".



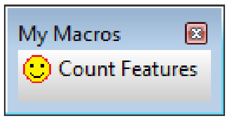
Finally, you can associate the image with the button definition you created in the Activate method of your Add-In. VB.Net uses a different type of object to represent bitmap data than VBA or VB6 did. Inventor's API expects the same type that VBA and VB6 uses, which is an *IPictureDisp* object. VB.Net uses the *Image* type, which is not compatible. When you access the picture as a resource it will be returned as an Image object. You'll need to convert it to an IPictureDisp object before using it as input to the AddButtonDefinition method. To do this you'll need to add references to two additional libraries. These are *stdole* and *Microsoft.VisualBasic.Compatibility*. These are both available on the .Net tab of the Add Reference dialog.

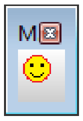
With these references available you have access to the types and functions you need. The code below demonstrates this. Notice how the bitmap in the resources is accessed using its name (My.Resources.Smiley).

```
' Convert the Image to a Picture.
Dim picture As stdole.IPictureDisp
picture = Microsoft.VisualBasic.Compatibility.VB6.ImageToIPictureDisp( _
            My.Resources.Smiley)

m_featureCountButtonDef = controlDefs.AddButtonDefinition( _
            "Count Features", _
            "AUAddInCountFeatures", _
            CommandTypesEnum.kQueryOnlyCmdType, _
            m_ClientID, _
            "Count the features in the active part.", _
            "Count Features", _
            picture )
```

Now the command shows up with an icon like that shown below. Text is displayed with the icon depending on the end-user setting they set using the context menu of the panel bar.





Creating Buttons within the Panel Bar
Creating a new toolbar and adding your commands to it, as shown above, is probably the easiest way to make your commands available but is not necessarily the most desirable. In many cases it's better to integrate your button in with Inventor's buttons. For example, if you write a command that is useful when working with assemblies it will be good to have it appear on the assembly panel bar with the rest of the assembly commands.

It's possible to position your commands anywhere within Inventor's user-interface; both in the classic and ribbon interfaces. The key to this in the classic interface is the CommandBar object. In inventor's user-interface, the panel bar, toolbars, menus, and context menu are all represented by CommandBar objects. To insert your button you need to find the existing command bar you want your button placed on.

One of the most common places to insert a button is the panel bar. The panel bar is just a container for a command bar so to display your command in the panel bar you need to find the correct command bar and insert your button into it. There is a default command bar defined for each environment. Here are the names of the more commonly used command bars, along with the internal name. The internal name is how you identify it in your program.

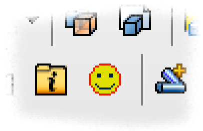
Assembly Panel,	AMxAssemblyPanelCmdBar
Assembly Sketch,	AMxAssemblySketchCmdBar
Drawing Views Panel,	DLxDrawingViewsPanelCmdBar
Drawing Sketch Panel,	DLxDrawingSketchCmdBar
Drawing Annotation Panel,	DLxDrawingAnnotationPanelCmdBar
Sheet Metal Features,	MBxSheetMetalFeatureCmdBar
Part Features,	PMxPartFeatureCmdBar
3D Sketch,	SCxSketch3dCmdBar
2D Sketch Panel,	PMxPartSketchCmdBar

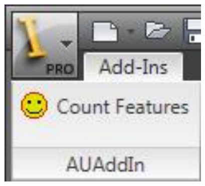
Here's a portion of code from the Activate method that demonstrates inserting a button into the part feature panel bar. This can replace the previous code that created the toolbar. Remember to change the version value in the registration portion of your Add-In so that Inventor will treat this as the first time your Add-In has been run.

```
If firstTime Then
    ' Get the part features command bar.
    Dim partCommandBar As Inventor.CommandBar
    partCommandBar = m_inventorApplication.UserInterfaceManager.CommandBars.Item( _
                     "PMxPartFeatureCmdBar")

    ' Add a button to the command bar, defaulting to the end position.
    partCommandBar.Controls.AddButton(m_featureCountButtonDef)
End If
```

This results in the part panel bar shown here or if you're using the ribbon interface you'll get a new panel in the Add-Ins tab when you have a part open, as shown below, on the right.





Cleaning up after your Add-In

When your Add-In is unloaded you should clean up and release references to Inventor objects you're still holding. This is an area that can get very confusing because of how .Net works. Rather than get into that discussion, here's a process that works. You only need to do this for global variables you've declared since other variables are automatically release when they go out of scope.

The Deactivate method of your Add-In exists for this purpose. Inventor calls the Deactivate method of your Add-In whenever it is being unloaded, which can happen when Inventor shuts down or the user unloads your Add-In using the Add-In Manager. The Add-In wizard created the code below for the Deactivate method. You can follow this pattern for other releasing other objects.

```
Public Sub Deactivate()
    ' Release objects.
    Marshal.ReleaseComObject(m_inventorApplication)
    m_inventorApplication = Nothing

    System.GC.WaitForPendingFinalizers()
    System.GC.Collect()
End Sub
```

Here's a modified version of the Deactivate after adding code to release the button.

```
Public Sub Deactivate()
    ' Release objects.
    Marshal.ReleaseComObject(m_inventorApplication)
    m_inventorApplication = Nothing

    Marshal.ReleaseComObject(m_featureCountButtonDef)
    m_featureCountButtonDef = Nothing

    System.GC.WaitForPendingFinalizers()
    System.GC.Collect()
End Sub
```

How do you debug an Add-In?

Debugging a COM component is an interesting problem. A COM component doesn't execute and run on its own but only when loaded and called by another program. In the case of an Add-In, the program loading and calling it is Inventor. To debug an Add-In you need Inventor to load it and make the necessary calls while you're able to monitor all of this in the debugging environment. Visual Studio supports this type of debugging but unfortunately this is a feature that's missing from the express edition of Visual Basic. However, we can work around this limitation by manually adding a few lines to one of the project files. If you're using Visual Studio, debugging is directly supported without this workaround. Debugging for both programming environments is described below.

Setting Up Visual Basic Express Debugging

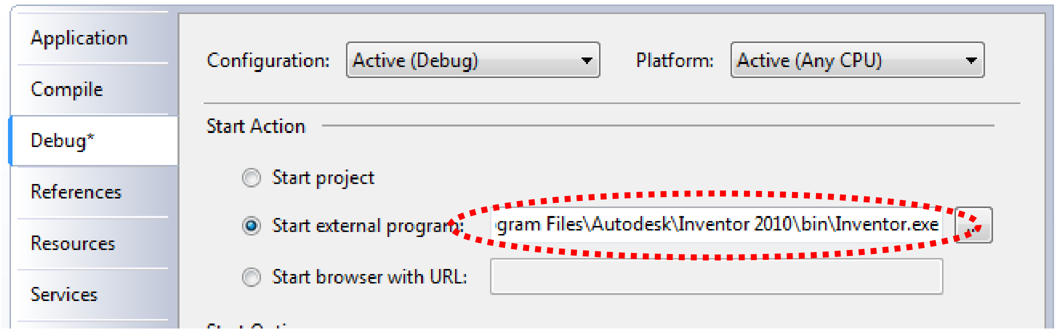
For my sample project, VB.Net created the file AUAddIn.vbproj.user to save some project settings. I added the two lines highlighted below.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <PublishUrlHistory>
    </PublishUrlHistory>
    <InstallUrlHistory>
    </InstallUrlHistory>
    <SupportUrlHistory>
    </SupportUrlHistory>
    <UpdateUrlHistory>
    </UpdateUrlHistory>
    <BootstrapperUrlHistory>
    </BootstrapperUrlHistory>
    <ErrorReportUrlHistory>
    </ErrorReportUrlHistory>
    <FallbackCulture>en-US</FallbackCulture>
    <VerifyUploadedFiles>true</VerifyUploadedFiles>
    <StartAction>Program</StartAction>
    <StartProgram>C:\Program Files\Autodesk\Inventor 2010\Bin\Inventor.exe</StartProgram>
  </PropertyGroup>
</Project>
```

The path to Inventor.exe will vary depending on where you installed Inventor and what version of Inventor you're using.

Setting Up Visual Studio VB.Net Debugging

Visual Studio provides additional debugging options to support debugging an Add-In. To set up debugging, open the Properties page for your project by running the Properties command, (the last command in the Project menu on the debug tab), select the **Start external program** option and browse to locate Inventor.exe.



Configuration: Active (Debug)
Platform: Active (Any CPU)

Start project
[✓ Start external program:] gram Files\Autodesk\Inventor 2010\bin\Inventor.exe
Start browser with URL:

Debugging your Add-In

Once debugging is setup, the process of debugging is the same whether you're suing Express or Visual Studio. To start debugging you use the **Start Debugging** command in the Debug menu. VB.Net will start Inventor and be in a state that you can debug the Add-In. Any break points you've inserted into your program will stop execution and allow you to step through your program. When a break point is hit you can examine values, step through code, make changes to code, and continue running.

How do you deploy an Add-In?

Now that you've got a working Add-In, how do you deploy it to other computers for others to use? This is another area where the Visual Basic Express Edition is missing functionality. The only built-in install capabilities the Express Editions come with is something called ClickOnce deployment. Unfortunately, this type of deployment doesn't have support for COM components, so you can't use it for an Add-In. Visual Studio does support the ability to create an installer but it is overly complex. Instead, I've found two other methods that work with both the Express and Visual Studio versions of VB.Net that I would recommend. These two methods are explained below. The first one described is the easiest method for you as the programmer, but not the easiest for the consumer of the Add-In. The second one takes a little more work on your side but provides much more power and flexibility and is the easiest for the Add-In user to use.

Method 1

The first and easiest way to deliver an Add-In is to copy the add-in to the target computer and register it. To copy the Add-In, copy the Add-In dll to the computer you want to run the add-In on using any means you're comfortable with, (email attachment, network drives, thumb drives, etc.). In my example, this would mean copying the AUAddin.dll to any location on the target computer.

To register the Add-In you need to run the regasm utility. Here's an example of its use. (The /codebase option is required in order to correctly register the Add-In.)

```
RegAsm.exe /codebase AUAddIn.dll
```

RegAsm.exe for Windows 32 bit is installed by the .Net Framework in the directory:

```
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727
```

For 64 bit Windows it is installed in:

```
C:\WINDOWS\Microsoft.NET\Framework64\v2.0.50727
```

You can also use RegAsm to unregister your Add-In. To make the register/unregister process easier you can create four small .bat files; 32 bit install, 64 bit install, 32 bit uninstall, and 64 uninstall. Here are the contents of the .bat files used to register the example Add-In. You'll need to change the name to match your Add-In.

Register32.bat

```
@echo off
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\RegAsm.exe /codebase AUAddIn.dll
PAUSE
```

Register64.bat

```
@echo off
C:\WINDOWS\Microsoft.NET\Framework64\v2.0.50727\RegAsm.exe /codebase AUAddIn.dll
PAUSE
```

Here are the contents of the two files to unregister the Add-In.

Unregister32.bat

```
@echo off
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\RegAsm.exe /unregister AUAddIn.dll
PAUSE
```

Unregister64.bat

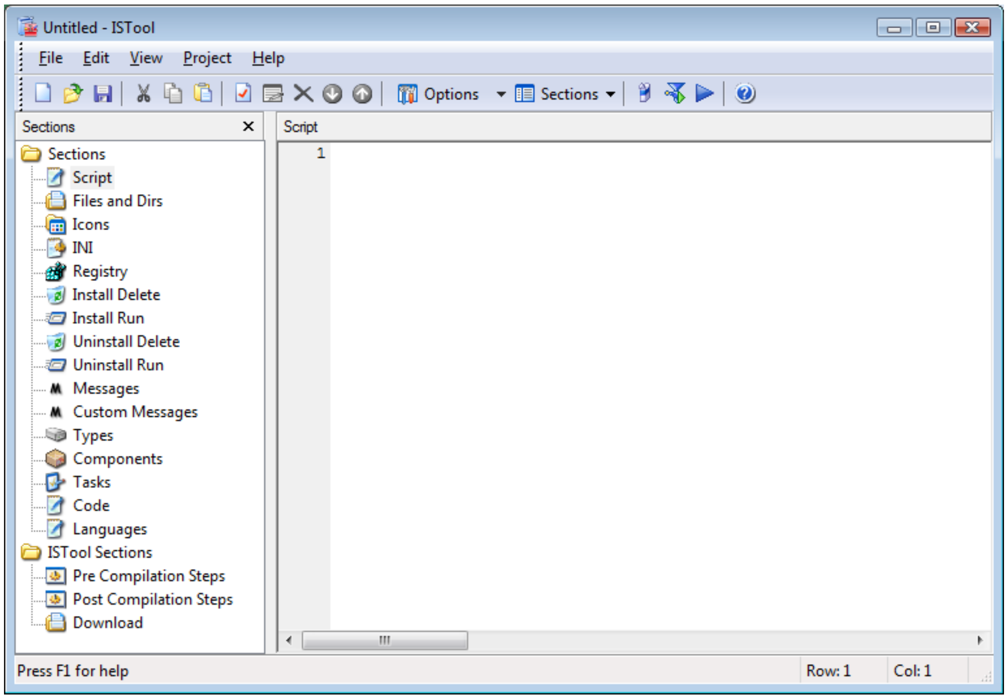
```
@echo off
C:\WINDOWS\Microsoft.NET\Framework64\v2.0.50727\RegAsm.exe /unregister AUAddIn.dll
PAUSE
```

Method 2

The second method relies on some third-party installation software. Even though Visual Studio allows you to create installers, the process is overly complex when all you actually need the installer to do is copy the file onto the system and run regasm. I also have yet to figure out how to use the Visual Studio installer to create a single installer that will correctly install an add-in for 32 or 64 bit Inventor. I've also tried another commercial installer, InstallShield Express, and had problems with it too. I've recently discovered a free installer that I've been using and have been very happy with it. It's called *Inno Setup* and is freely available on the web at: http://www.jrsoftware.org/isinfo.php. Below are the step-by-step instructions to creating a setup using Inno Setup.

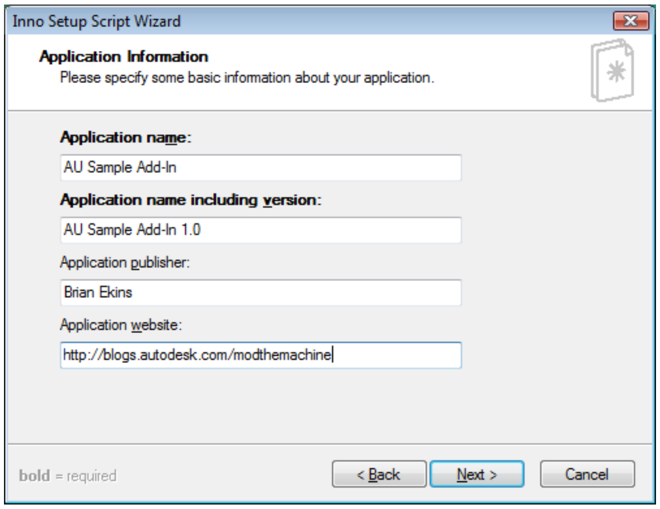
1. Download the QuickStart pack. When you go to the Inno Setup site, you'll see there are several download options. You should download the Unicode version of the QuickStart pack and take the default options for what to install. The QuickStart pack includes Inno Setup plus some additional utilities to make using it easier.

2. Run ISTool. This is one of the utilities delivered as part of the QuickStart pack and provides a user-interface to Inno Setup. You should see something similar to that shown below.



3. Run the **New** command, which will display the Inno Setup Script Wizard.

4. Enter the information on the first page of the wizard. The picture below shows my entries for this sample Add-In.



Inno Setup Script Wizard

Application Information
Please specify some basic information about your application.

Application name: AU Sample Add-In

Application name including version: AU Sample Add-In 1.0

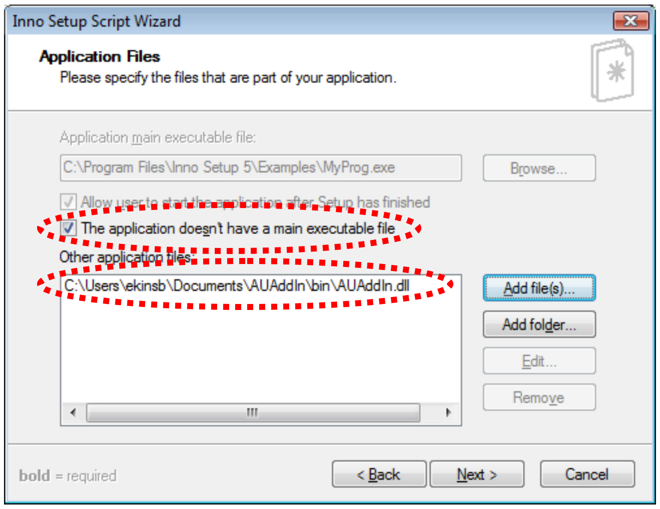
Application publisher: Brian Ekins

Application website: http://blogs.autodesk.com/modthemachine

bold = required

5. Take the default values for the **Application Folder** page of the wizard.

6. For the **Application Files** page, check the box for "The application doesn't have a main executable file" and click the **Add files(s)…** button to browse and find the Add-In dll, as shown below.



Inno Setup Script Wizard

Application Files
Please specify the files that are part of your application.

Application main executable file: C:\Program Files\Inno Setup 5\Examples\MyProg.exe

[✓] Allow user to start the application after Setup has finished

[✓] The application doesn't have a main executable file

Other application files:
C:\Users\ekinsb\Documents\AUAddIn\bin\AUAddIn.dll

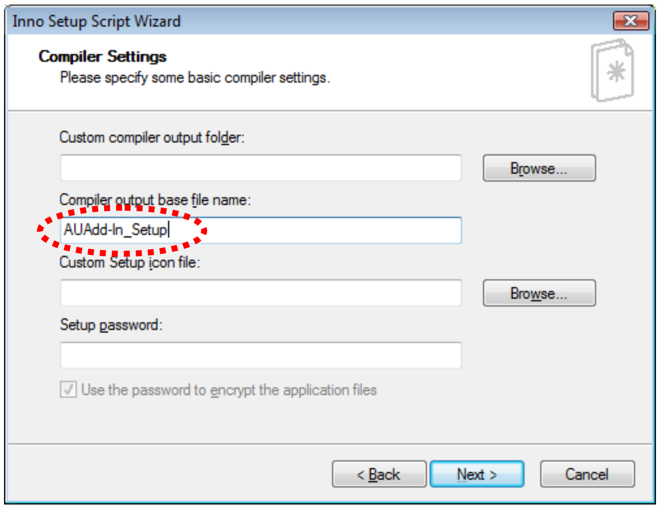
bold = required

7. Take the default settings for the **Application Icons** page of the wizard.

8. Take the default settings for the **Application Documenation** page of the wizard.

9. On the **Setup Languages** page, choose any languages you want to support. The languages chosen will only affect the installer, not your Add-In.

10. On the **Compiler Settings** page, change the "Compiler output base file name:" to include the name of your Add-In.



Custom compiler output folder:

Compiler output base file name: AUAdd-In_Setup

Custom Setup icon file:

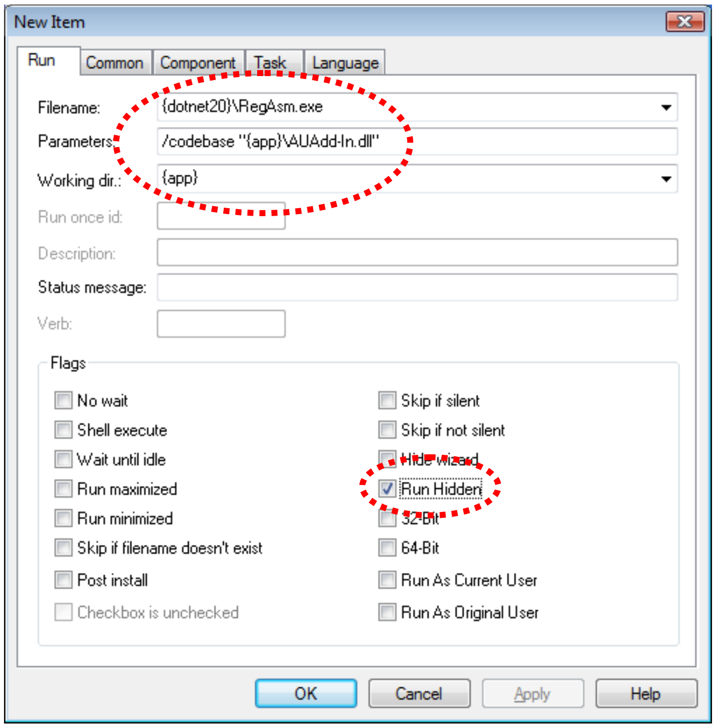
Setup password:

☑ Use the password to encrypt the application files

11. Take the default settings for the **Inno Setup Preprocessor** page of the wizard.

12. Click **Finish** which will finish the wizard and take you back to the ISTool interface. The setup project you're creating is defined in a script file. The wizard and ISTool provide a user-interface for editing the script. You can compile the setup now and it will create a setup file that will copy the Add-In dll to the target machine, however the Add-In won't work because you haven't configured your setup to perform the registration required by your Add-In. You still need to add some instructions to the setup for it to do the registration.

13. In ISTool, click the **Install Run** section in the Sections list and then right-click anywhere in the right-hand area of ISTool and choose **New Item…** from the context menu, which will display the dialog below. Edit the **Filename**, **Parameters**, and **Working dir** fields as shown below. (It's curly braces { } around "dotnet20" and "app"). Also, check the **Run Hidden** flag in the Flags section. Click OK to save the changes and dismiss the dialog.

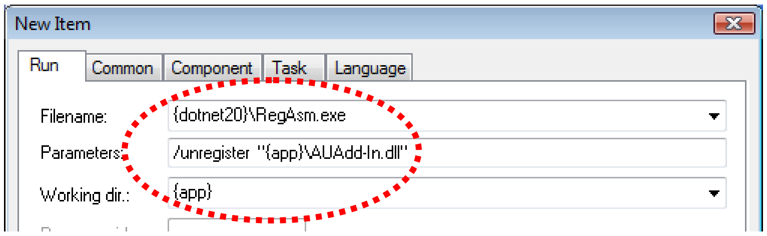


Filename: {dotnet20}\RegAsm.exe
Parameters: /codebase "{app}\AUAdd-In.dll"
Working dir.: {app}
Run once id:
Description:
Status message:
Verb:

Flags:
- [ ] No wait
- [ ] Shell execute
- [ ] Wait until idle
- [ ] Run maximized
- [ ] Run minimized
- [ ] Skip if filename doesn't exist
- [ ] Post install
- [ ] Checkbox is unchecked
- [ ] Skip if silent
- [ ] Skip if not silent
- [ ] Hide wizard
- [✓] Run Hidden
- [ ] 32-Bit
- [ ] 64-Bit
- [ ] Run As Current User
- [ ] Run As Original User

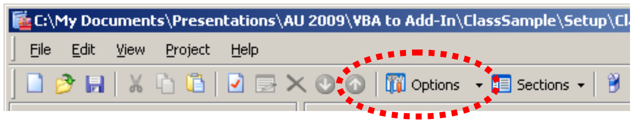
The changes you made will cause RegAsm to be run for your Add-In during the installation process. The {dotnet20} parameter will use either the 32 or 64 bit version of RegAsm depending on what version of Windows the installer is running on. This allows this single installer to register correctly for both 32 and 64 bit systems.

14. The installer also needs to remove the registry entries when the Add-In is uninstalled. To do this, click the **Uninstall Run** section and create a new item, like you did in the last step. Enter everything the same as you did in the previous step except replace /codebase with /unregister, as shown below.

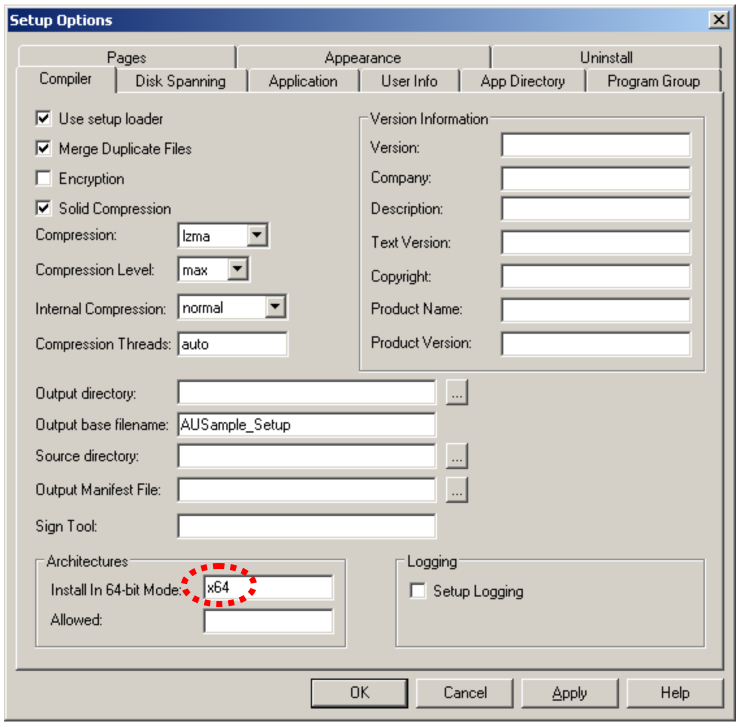


Filename: {dotnet20}\RegAsm.exe
Parameters: /unregister "{app}\AUAdd-In.dll"
Working dir.: {app}

15. One more setting needs to be made in order for your add-in to be correctly installed for both 32 and 64 bit operating systems. Run the **Options** command, as shown below.


C:\My Documents\Presentations\AU 2009\VBA to Add-In\ClassSample\Setup\Cl
File
Edit
View
Project
Help
Options
Sections

This will display the Setup Options dialog. On the Compiler tab, edit the “Install In 64-bit Mode” option to be “x64” as shown below.



- [✓] Use setup loader
- [✓] Merge Duplicate Files
- [ ] Encryption
- [✓] Solid Compression

Compression:	lzma
Compression Level:	max
Internal Compression:	normal
Compression Threads:	auto
Output directory:	
Output base filename:	AUSample_Setup
Source directory:	
Output Manifest File:	
Sign Tool:	
Install In 64-bit Mode:	x64
Allowed:	

- [ ] Setup Logging

16. Run the Compile Setup command to create the setup file. You can choose to test it immediately after compiling if you want. This creates a standard setup that makes distributing your Add-In better than the .bat file approach because it’s simple for the end-user to install and can be uninstalled using Windows control panel.