Autodesk University 2008

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

Brian Ekins – Autodesk

DE301-1 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 Express. 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 been programming Inventor for very long you've likely heard about Add-Ins. The goal of this paper is to help you understand what an Add-In is and determine if it's something that better meets your needs than using Inventor's VBA. There are several questions we'll address:

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 the other program to talk to it. In the case of an 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 starts and interacts with each one. The interaction between Inventor and the Add-In at this point is minimal but consists of 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 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 that button represents.

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 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 an Inventor command and a well written Add-In command.

2. Alternative to document automatic macros

Inventor supports the ability to create a 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 and Add-In is a better choice.

3. **Can use new languages**
   An Add-In can be created using any language that supports the creation of a COM component. This provides 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 as the programming language.

4. **Can fully integrate into Inventor's user-interface**
   An Add-In can provide a much richer user-interface than what you can do with VBA. For example, the languages you create an Add-In with have much better dialog creation tools. The Add-In can 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 run 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**
   Source code of Add-Ins is easier to manage than VBA programs. VB.Net source code is saved as ASCII files which allows you to easily search them. 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 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? Probably not. Add-Ins are 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 likely still 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?

We'll look at creating an Add-In using Visual Basic 2008 Express Edition. This is a free version of VB.Net and can be downloaded from Microsoft's website. Microsoft also provides a lot of training material that is free and much of this is targeted at the new programmer. It would be good to work through some of these tutorials to familiarize yourself with the VB Express development environment.

Of course when you get something for free you expect some limitations. The complete feature set of the Express editions compared to the other versions can be seen on Microsoft's website. For Add-In development you get everything you'll likely need in the Express edition with the exception of two things, which I'll show you how to work around. You can always upgrade later to the Standard Edition for $299 if you decide you need more functionality.

Installing the Add-In Template

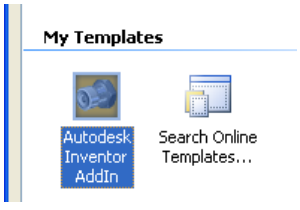
Included with the Inventor SDK is a *template* used to create Add-Ins using VB and C#. At the time Inventor 2008 was released VB 2005 Express Edition was the current release, so installing the template does not install it for VB 2008 Express. You can easily work around this issue using the steps below.

1. Get the template file, *VBInventorAddInTemplate.zip*. You can get this file as part of the packaged set of files for this AU session or 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 file as-is): My Documents\Visual Studio 2008\Templates\ProjectTemplates

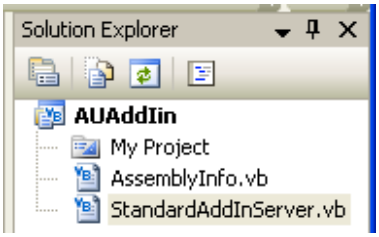
Normally you just run the InventorWizards.msi file from the directory shown below to install the template but the workaround above is to support VB 2008 Express with Inventor 2009.

Program Files\Autodesk\Inventor 2008\SDK\Tools\Developers\Wizards

Once the template is installed you create an Add-In by starting VB 2008 Express (which I'll just refer to as VB.Net for the rest of this paper). Within VB.Net create a new project using the **New Project** command in the File menu. The New Project dialog is shown, which displays the templates that are available to create various types of projects. Select the "Autodesk Inventor AddIn" template as shown to the right. Specify a name that makes sense for your project, (I used "AUAddIn"), and click OK.

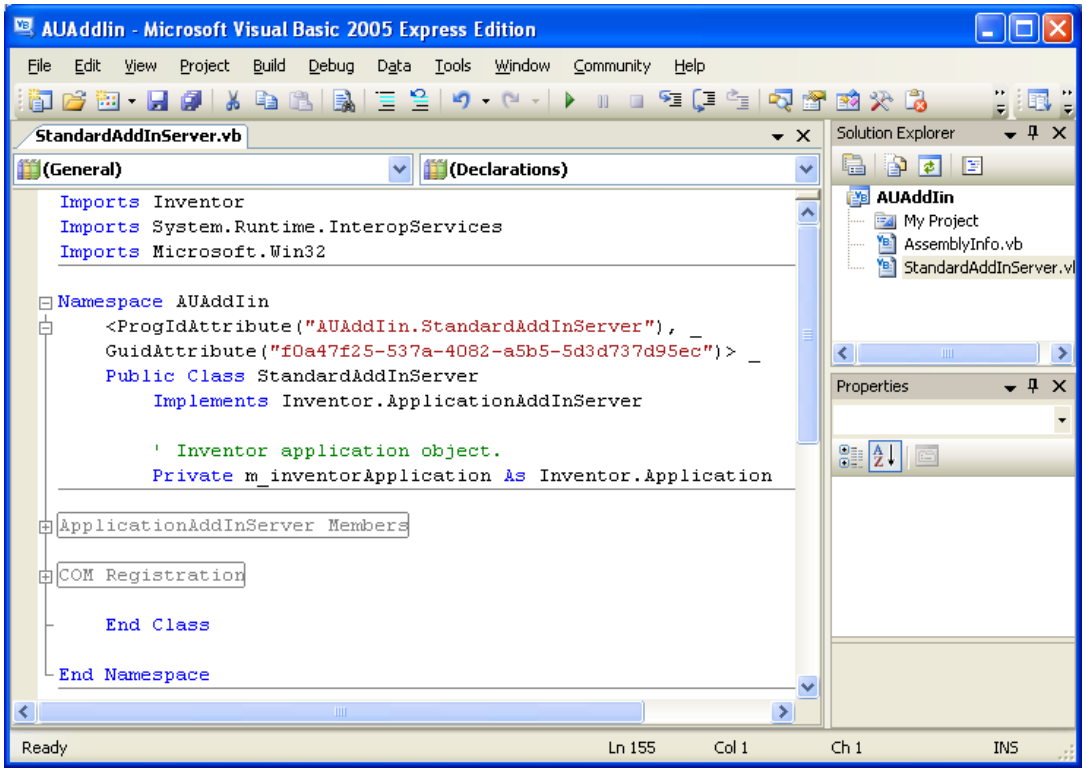


Congratulations, you've just created an Add-In. Click the **Build *AUAddIn*** (or the name or your Add-In) command in the Build menu. Once it's finished building, start Inventor and look in the Add-In Manager. You should see your Add-In in the list. It's not very exciting at this point because the Add-In 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 that VB uses 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.



At first glance it looks like there's not very much code, but most of it is hidden within two 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 Client ID. The client ID is the big number in the GuidAttribute line; "f0a47f25-537a-4082-a5b5-5d3d737d95ec" in my case. This is the unique identifier for your Add-In (known as the Client ID). You'll see later how this is used when creating things associated with the Add-In.

The other important item is the declaration of the variable `m_inventorApplication`. It's only when programming in VBA that you have access to the ThisApplication property. An Add-In needs to get the

Application object in some other way. The m_inventorApplication variable is used to save the reference to the 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 alone 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 define which version of Inventor your Add-In will be loaded for. Notice that all but one of these lines is commented out. They illustrate the various options available. In this example, the Add-In will load for all versions of Inventor later than version 12 (Inventor 2008), so it will load for Inventor 2009 and beyond. 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", "12..")
        '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 not commonly used. You will use the Activate and Deactivate methods.

The Activate method is what Inventor calls when it starts up the Add-In. This method has two arguments that Inventor uses to pass information to the Add-In. The first argument is an ApplicationAddInSite object. This object supports the Application property which the Add-In uses to get the Inventor Application object and assigns it to the **`m_inventorApplication`** member variable.

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 references to all 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?

In converting VBA code into an Add-In there are two things to consider. First, where does the 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 between them that you’ll need to be aware of. The issues you’ll most commonly run into are discussed below.

Converting VBA Code

Converting VBA code to VB.Net is a copy and paste process. You can 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 a lot of code will require some 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 most common issues 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 to use, is primarily your personal preference.

   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
```

   The reason it doesn't work in VB.Net is because of the use of the kJoinOperation constant. In VB.Net you must fully qualify its use by also specifying the enumeration name as shown below. These are easy to catch and fix since VB.Net identifies them as errors and IntelliSense does most of the work in creating the fully qualified name.

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

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 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. For arguments not specified it automatically sets them 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 to function correctly.

4. **Arrays have a lower bound of 0 (zero).** I think this is probably 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 bound 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 specifies 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 brackets).

```
' 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 type isn't supported in VB.Net. If you have programs that use this type just change those to use the 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 the 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 are reported as that variable not being 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 is now within the Sub End Sub block, 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. **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. The Call statement is still supported but no longer needed.

   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.

Cool stuff in VB.Net

As we see above, 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.

4. Debugging is better.

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 coding with any of the other .Net languages.

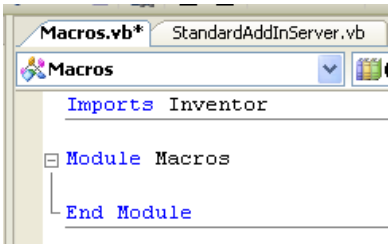
6. Incrementing a variable. There’s now some simpler syntax for incrementing the value of a variable.

```
' 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 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 and adding the Imports line for the Inventor library.

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
    Set 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. 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 has to be rebuilt. Recreating your dialogs isn’t necessarily a bad thing since VB.Net has better dialog tools and supports a richer set of controls than were available in VBA.

You create a dialog as a Visual Basic form and display it in response to the button definition's OnExecute event. The code below illustrates responding to the 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.

```
#Region "hWnd Wrapper Class"
' This class is used to wrap a Win32 hWnd as a .Net IWind32Window class.
' This is primarily 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. Add-Ins don't provide such an easy interface but do provide additional options and flexibility. For now I'll keep it simple by looking at the minimum work needed to create a button to execute the sub in your Add-In.

The first step is to create a ButtonDefinition object to represent your command. The ButtonDefinition object defines what your button looks 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 references the ButtonDefinition. You create the ButtonDefinition object every time the Add-In is started. However, you only create the

CommandBarControl the very first time the Add-In is run (when the firstTime argument is True). You only create it the first time because Inventor remembers its position after that.

Below is some sample code that demonstrates all of this.

```
' Declare member variables.
Private m_inventorApplication As Inventor.Application
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

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

    m_featureCountButtonDef = controlDefs.AddButtonDefinition( _
            "Count Features", _
            "AUAddInCountFeatures", _
            CommandTypesEnum.kQueryOnlyCmdType, _
            "{f0a47f25-537a-4082-a5b5-5d3d737d95ec}", _
            "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",, _
                                     "{f0a47f25-537a-4082-a5b5-5d3d737d95ec}")
        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.

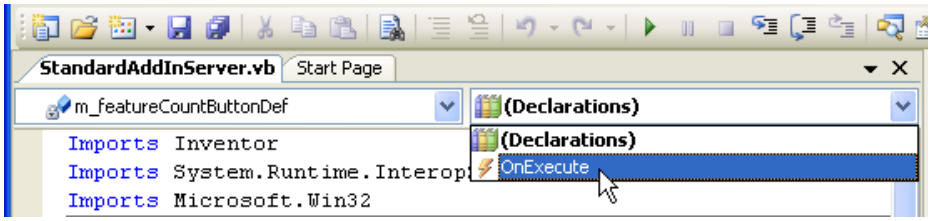
1. The first argument is the display name. This is used for the text on the button.
2. The second argument is the internal name. This is a unique identifier you define for this particular control definition and must be unique with respect to all other control definitions in Inventor.
3. The third argument categorizes this command. The two most common categories are kQueryOnlyCmdType and kShapeEditCmdType. The first is for a command that just queries and the second is for a command that modifies geometry. There are others that are described in online help.
4. The fourth argument is the client ID of your add-in. Note that it's enclosed within brackets.
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 that 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.



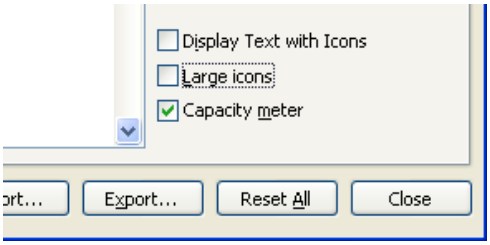
The OnExecute event handler code is inserted into your project. The code below illustrates the code you write to call the FeatureCount sub whenever the button is clicked. The Inventor Application object is passed as the single argument.

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

Creating Icons for your Commands

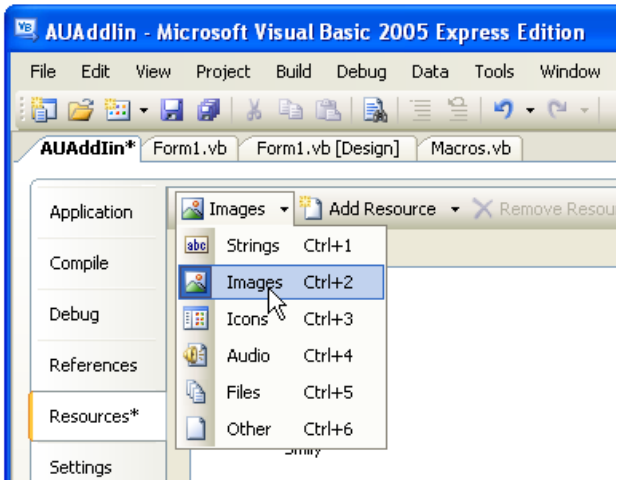
It’s also good to have an icon in addition to the command name to minimize the size of the button and make the button consistent with the rest of the Inventor commands. Here are the steps for creating and using icons for your buttons. Visual Basic 2008 Express does not provide a built-in tool to create 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*

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 specifically create the large icon.

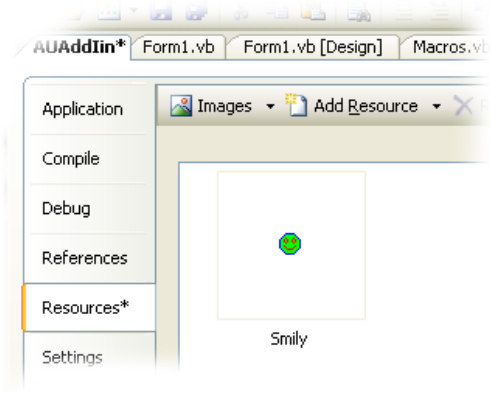


Icon design is somewhat of an art and can consume a lot of time to create something that looks good and represents the associated command. I’ve had my best luck designing icons by finding a command in Inventor whose icon has elements similar to the command I’m writing. I copy the icon by doing a screen copy and then edit it to get the desired result.

For this example I used the Paint program delivered with Windows to create a 16x16 icon and saved it as a .bmp file. A .bmp file can be imported into a VB 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 below 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. Select your existing .bmp or .ico file to add it to the project. Finally, assign a logical name to the resource. You can see below that I've named my image "Smily".



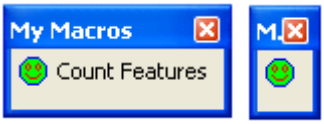
Finally, you can associate the image with the button definition you created in the Activate method of your Add-In. .Net uses a different type of object to represent bitmap data than VBA or VB6 did. Inventor's API uses the type that VBA and VB6 supported which is an *IPictureDisp* object. .Net uses the *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 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 to create a button definition with an icon. 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.Smily)

m_featureCountButtonDef = controlDefs.AddButtonDefinition( _
            "Count Features", _
            "AUAddInCountFeatures", _
            CommandTypesEnum.kQueryOnlyCmdType, _
            "{f0a47f25-537a-4082-a5b5-5d3d737d95ec}", _
            "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. The key to this is the CommandBar object. To insert your button into Inventor's user-interface you need to find the existing command bar you want your button placed on. In inventor the panel bar, toolbars, menus, and context menu are all represented by CommandBar objects and all of them are accessible through the API.

One of the most common places to insert a button is the panel bar. The panel bar displays a command bar so you need to find the correct command bar to insert your button into to have it display in the panel bar. 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.

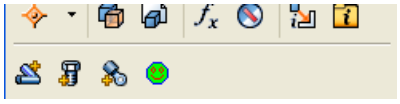
Assembly Panel, AMxAssemblyPanelCmdBar
Assembly Sketch, AMxAssemblySketchCmdBar
Drawing Views Panel, DLxDrawingViewsPanelCmdBar
Drawing Sketch Panel, DLxDrawingSketchCmdBar
Drawing Annotation Panel, DLxDrawingAnnotationPanelCmdBar
Sheet Metal Features, MBxSheetMetalFeatureCmdBar
Part Features, PMxPartFeatureCmdBar
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.

```
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.



How do you debug an Add-In?

Debugging a COM component is an interesting problem. A COM component isn't executed 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 the Add-In 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.

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 2009\Bin\Inventor.exe</StartProgram>
  </PropertyGroup>
</Project>
```

The portion of this file that can change is the path to Inventor.exe. It needs to point to the version of Inventor you're developing against. With this file in place you can 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 insert into your program will stop execution and allow you to step through your program. You can 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.

The easiest way to deliver a Visual Basic Express Edition Add-In is to copy the add-in to the target computer and register it. To copy the Add-In, copy the dll that's created when you build the Add-In project to the target computer. In my example this is the AUAddin.dll in the bin directory where the Add-In was saved. This file can be copied 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 Visual Studio 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 this 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 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
```