AUTODESK UNIVERSITY

SD224078, Version 2

Creating Add-Ins for Inventor

Brian Ekins
Ekins Solutions, LLC
brian@EkinsSolutions.com

Learning Objectives

- Discover when an add-in is the best option.
- Learn how to convert existing VBA and iLogic code into an add-in command.
- Learn how to debug an add-in.
- Learn how to package and distribute your add-in.

Description

While Inventor software's VBA and iLogic provide a convenient and easy way to add custom functionality to Inventor, there are limitations to what you can do with VBA macros and iLogic rules. This class will introduce you to VB.NET and Inventor add-ins and the pros and cons of add-ins compared to VBA and iLogic. In this class, you'll learn the technical differences between VBA, iLogic, add-ins, and external EXEs, and some of the reasons you would choose one over the other. The class will cover how to create an add-in using an easier-to-use custom VB.NET template. We'll look at the differences between VBA code and VB add-in code and how to convert your existing VBA macros and iLogic rules into add-in commands. We'll also cover how to debug your add-in. Finally, we'll look at how to package and distribute your add-in to make it easy for others to use. This is an intermediate level class and you should have a basic understanding of the Inventor API.

Speaker(s)

CAD has been a passion of mine ever since I saw a demonstration of an Applicon system during a high school field trip back in 1977. My experience in the industry since then has been varied. I've developed and taught training classes, both on modeling and programming. I've worked as an Application Engineer where I specialized in complex modeling and customization of the software and I demonstrated the software and performed benchmarks to show customers that the software could do what they need. When Intergraph began developing Solid Edge I moved from the role of a modeler and API user to an API designer. I then moved to Autodesk where I designed the API for Inventor and most recently, I've been designing the API for Fusion 360. Now I run my own consulting/contracting business where I help Inventor and Fusion 360 users make Inventor and Fusion 360 into a customized tool specific to their needs. Contact me if you need help with your use of the software. https://EkinsSolutions.com

Table of Contents

Introduction

Before writing this paper, I polled the people who will be attending the AU class and asked them a few questions. Out of the respondents, I found out that a large percentage are not currently writing programs for Inventor so although this topic is a more advanced Inventor API topic, I want to give those of you that are new to Inventor programming some background and other resources to help you get started and to be able to understand the rest of the paper.

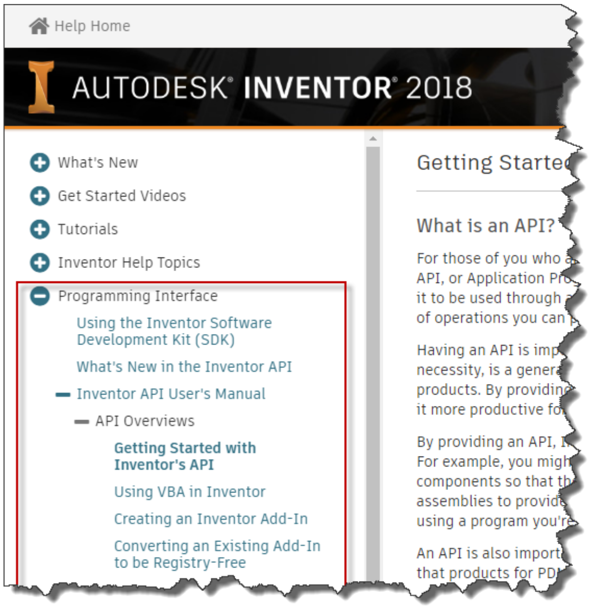
What is an API?

An “Application Programming Interface” or “API” for short, is something that an application supports that allows you to control it using a program. Everyone is familiar with graphical user interfaces. For example, in Inventor you have the graphics windows where you view your model and you have the ribbon and context menu where you can find specific commands and run them. When a command is running, you interact with the dialog to specify various options and select items in the graphics windows to specify geometric input and then you complete the command to make changes to the model.

In a graphical user interface, you specify that you want to perform an operation by starting the appropriate command. You then provide the information that the command requires by selecting options in the dialog and choosing geometry, and then you finish the operation by completing the command.

A programming interface is conceptually the same but instead of running a command and choosing options in a dialog or selecting geometry in the graphics window you instead call a programming function that is the equivalent of the command. You provide the same kind of input to the function that the command requires. For example, you provide values to indicate the values of the settings and you provide the geometry that you obtained using other functions in the API. When the function executes, it results in the same result as the command. In fact, internally the command and the API function end up calling the same internal function to create the desired result.

The rest of this paper assumes you have a basic understanding of Inventor’s API, but if you don’t you can get some basic information in the Inventor Help where’s there’s a section specifically for the API, as shown to the right.


Help Home
AUTODESK® INVENTOR® 2018
What's New
Get Started Videos
Tutorials
Inventor Help Topics
Programming Interface
Using the Inventor Software Development Kit (SDK)
What's New in the Inventor API
Inventor API User's Manual
API Overviews
Getting Started with Inventor's API
Using VBA in Inventor
Creating an Inventor Add-In
Converting an Existing Add-In to be Registry-Free

Getting Starte

What is an API?

For those of you who a
API, or Application Pro
it to be used through a
of operations you can p

Having an API is imp
necessity, is a genera
products. By providin
it more productive fo

By providing an API, I
For example, you migh
components so that th
assemblies to provid
using a program you'r

An API is also import
that products for PD

Tools for Programming Inventor

Before digging into the details of add-ins, let's first look at the various ways you can access Inventor's API and the pros and cons of each. Even before that, let me say that Inventor exposes its API using some Microsoft technology called COM (Component Object Model). It's not important to understand what this is but its good to understand that it is exposed using this generic technology. Because of this, it's possible to use any programming language that supports COM. It's also useful because this is also how many other applications expose their API's. For example, Word, Excel, Visio, Solidworks, Solid Edge, and many others use this same technology. This means that much of what you learn programming one is transferrable when programming another.

VBA (Visual Basic for Applications)

VBA is built into Inventor and provides a very good development environment for writing programs that work with the Inventor API. VBA was designed to program COM interfaces and because of that, it does provide the best integration.

Pros:

- Free and included with Inventor
- Good editor with Intellisense and code completion
- The best debugging
- The best object browser
- Can create custom dialogs
- Most of the online help samples are VBA code
- In context API help

Cons:

- Based on old VB 6 technology
- Limited dialog creation
- Difficult to incorporate into Inventor user interface
- Cannot use Apprentice
- Difficult to respond to events
- Practically, it's limited to macro creation
- Usually executed from the Macros dialog
- Difficult to share with others

iLogic

iLogic is a tool that's delivered with Inventor that was originally designed to allow easy configuration of parts. You can create "rules", which are basically programs that modify parameters and the suppression state of features to result in part configuration. You can easily define the inputs for a part and iLogic creates a form to let the user edit those values and then modifies the part. To use some of this basic functionality you don't even need to write any code.

For this basic type of functionality, it's difficult to beat iLogic. To create more complicated rules, iLogic provides a code editor and supports some iLogic specific functions that are intended to make it simpler to use than the Inventor API. However, it's also possible to access the complete Inventor API from within an iLogic rule. What's happened is that people started writing simple iLogic rules but needed more functionality and started incorporating the Inventor API into their rules. Many people are now writing pure Inventor API programs as iLogic rules. This works but it also means you must deal with all the cons of iLogic. The pros and cons list below is with respect to using iLogic as a general programming environment for the Inventor API.

Pros:

- Free and included with Inventor
- Uses the new .NET based languages and libraries
- Easy to listen to and respond to events
- Simplified tools for configuring parts

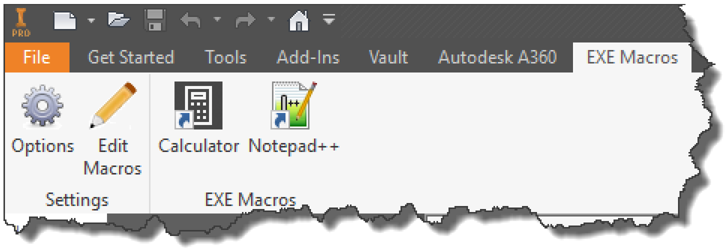
Cons:

- Poor editing
- No interactive debugging
- No object browser
- Cannot use Apprentice
- Limited to macro creation or responding to iLogic supported events
- Usually executed from the iLogic browser
- Difficult to share with others

EXE Macros

This is a very old concept that I've updated to make more user-friendly. It's always been possible to write an executable (EXE) program that can use the Inventor API. You create the EXE and then from Windows Explorer run the EXE and it connects to and drives Inventor using the API. The big advantage with an EXE macro is that you can use Visual Studio to write it. The problem with creating EXE's is that you need to run them from Windows Explorer. You can't easily add new buttons into the user interface.

My new "Nifty EXE Macro" capability lets you create EXE's in the same way they've always been created but then lets you easily create buttons in the Inventor user interface to execute them. You can define the icon for the button, the tooltip, the description, and even the picture that's displayed for the extended tooltip. And it's not limited to EXE's that use the Inventor API, but you can create buttons that will run any EXE. For example, you can create a button to open the Calculator program or Notepad, as shown below. For more information about EXE Macros, visit my website at https://EkinsSolutions.com.



Writing an EXE is much easier than writing an add-in but because it uses the same technology (Visual Studio), it is a great way to start using the API and to eventually creating add-ins.

Pros:

- Uses Visual Studio
- Uses the new .NET based languages and libraries
- Easy to add a button into user-interface
- Can use Apprentice
- Can easily be shared with others
- Can use any language that supports COM

Cons:

- Must install and license Visual Studio
- Practically, it's limited to macro creation
- Can use events but not good for most Inventor events
- Slower performance because of running out-of-process

Add-Ins

Now, we're finally getting to the main topic of Add-Ins. All the programming methods described so far have full access to Inventor's API. Some provide better or easier access to certain parts of the API, (primarily events and Apprentice), but all of them are using the same Inventor API and as a result, can do the same things inside Inventor. The big difference between them from the developer's perspective is how easy it is to write that code. The big difference between them from the end user's perspective is how easy it is to access the functionality that the program provides and if it works as expected.

The big thing that an add-in provides is the ability to provide easy, intuitive access to the commands it provides. Add-Ins do this by being started automatically by Inventor when Inventor is started. As part of the start-up process, an add-in adds buttons for its commands into the Inventor user interface. The add-in continues to run throughout the entire Inventor session but it's just sitting quietly in the background waiting to respond when one of its command buttons is clicked by the user. Once a command is clicked then there's not much, if any, difference between the code that would be written for iLogic or an EXE Macro.

To the end-user, commands provided through an add-in appear to be standard Inventor commands. For example, iLogic is an add-in and so is Tube and Pipe, Cable and Harness, and Frame Generator. All these programs use the Inventor API to provide access to their commands and for their interaction with Inventor and they appear to be a core part of Inventor.

Because they're loaded when Inventor starts and continue to run in the background, they're also the best solution when you have an event-driven program. When they're loaded they can connect to the events of interest and then react accordingly when Inventor calls them when that event is triggered.

Like an EXE, you're compiling your add-in to create code that is executed at runtime. However, with an add-in, you're creating a DLL which is loaded into the Inventor process and run.

Pros:

- Uses Visual Studio
- Uses the new .NET based languages and libraries
- Best at adding commands anywhere in the user interface (ribbon, context menu, toolbars, etc.)
- Best at being able to use all the events supported by the Inventor API
- Can be easily shared with others
- Can use any language that supports COM

Cons:

- Must install and license Visual Studio
- More complicated to develop
- Cannot use Apprentice

Visual Studio

For creating an EXE or an add-in you need to install and use Visual Studio. Visual Studio provides the best code editing experience of any of the Inventor programming solutions with the following features:

Code Editing
Visual Studio is easily the best development environment of the programming environments listed above. The thing that you'll quickly appreciate and deeply miss in most other development environments is what Microsoft calls "Intellisense". It's a code-completion aid that shows you viable options and enters the code for you.

Interactive Error Handling
As you write your program, Visual Studio is constantly checking for errors and displays them by highlighting your code and providing more detailed information in the "Error List" window.

Debugging
The debugging isn't quite as good as VBA but is still very good.

Object Browser
Here again, the Object Browser isn't quite as good as VBA, but Visual Studio still provides an Object Browser to let you view the objects, methods, and properties provided by Inventor's API.

Form Development
Visual Studio provides the best tools for creating custom dialogs. You can develop dialogs using Windows Forms or the newer Windows Presentation Foundation (WPF).

Visual Studio is one of Microsoft's Integrated Development Environments (IDE). There are different versions of Visual Studio and they all support several different languages (Visual Basic, C#, C++, etc.). Here's a brief description of each type of Visual Studio to help you understand what's available and what might be the best fit for you.

Visual Studio Professional (https://visualstudio.microsoft.com/vs/professional/)

This is the version that most professional developers are using. It supports everything that's needed to create an add-in. It's purchased from Microsoft and is licensed so you can sell the software that you write. The full retail price for a license of Visual Studio Professional is $499.

Visual Studio Community (https://visualstudio.microsoft.com/vs/community/)

This is a version that was introduced by Microsoft a few years ago. It has the same capabilities as Visual Studio Professional, but it is free. However, with free also comes a limited license. Here's who can use it for free as quoted straight from Microsoft's website.

> **Individuals** - Any individual developer can use Visual Studio Community to create their own free or paid apps.
>
> **Organizations –** An unlimited number of users within an organization can use Visual Studio Community for the following scenarios: in a classroom learning environment, for academic research, or for contributing to open source projects.
>
> For all other usage scenarios:
> In non-enterprise organizations, up to five users can use Visual Studio Community. In enterprise organizations (meaning those with >250 PCs or >$1 Million US Dollars in annual revenue), no use is permitted beyond the open source, academic research, and classroom learning environment scenarios described above.

Visual Studio Express (https://visualstudio.microsoft.com/vs/express/)

Visual Studio Express is also free but instead of having a restricted license, it has restricted capabilities. This means anyone, including someone that is part of an enterprise organization, can install and use Visual Studio Express to create free or paid apps. If you don't qualify for the Community version, Express is a great option. However, Microsoft is saying that Visual Studio Express 2017 will be the last version and then will be discontinued. After that, you'll need to use either the Community or Professional version.

Visual Studio Express does have a few limitations and unfortunately, these limitations do affect creating add-ins, but they can be worked around so it is possible to still create add-ins. The limitations and workarounds will be described in this paper.

Visual Studio Code (https://code.visualstudio.com/)

You might hear about one other Visual Studio variant called Visual Studio Code. It's a new open source multi-platform development environment. It's most commonly used for web development but can be used for other types of development too. However, it currently doesn't support Visual Basic or the capabilities needed to write an Inventor add-in.

Creating an Add-In

The add-in code that interacts with Inventor isn't really any different than the code that would be written using one of the other ways of accessing the API that were described earlier. However, there is some additional structure around an add-in that allows it to be found and loaded by Inventor. It's all this additional structure and code that can make developing an add-in a bit daunting, especially for a newer programmer. I would recommend that you start by writing EXE Macro programs, which can be easily converted into an add-in later, if you decide that will be better.

Templates

To help create all the required code structure, you can use an add-in template to create the initial add-in code. There are two different templates, as described below. The rest of this paper assumes that you're using the second option, the Nifty Add-In Template.

Standard Add-In Template

To get access to the Inventor supplied add-in template you need to install the **Inventor SDK**. When you install Inventor, an installer for the Inventor SDK (Software Development Kit) is also installed but then you need to manually run that installer to install the SDK. The installer for the SDK is installed in (where XXXX is the version):

C:\Users\Public\Documents\Autodesk\Inventor XXXX\SDK

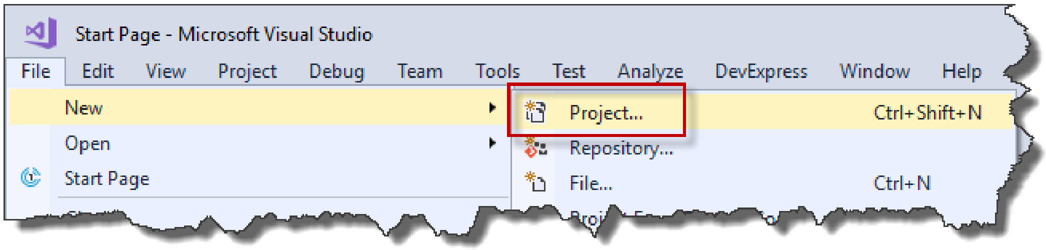
Inside that folder, run "developertools.msi". This will install the add-in template for Visual Basic, C#, and C++ and will also create the DeveloperTools folder in the same directory above. The DeveloperTools folder contains many samples and tools that are useful when developing for Inventor.

Nifty Add-In Template

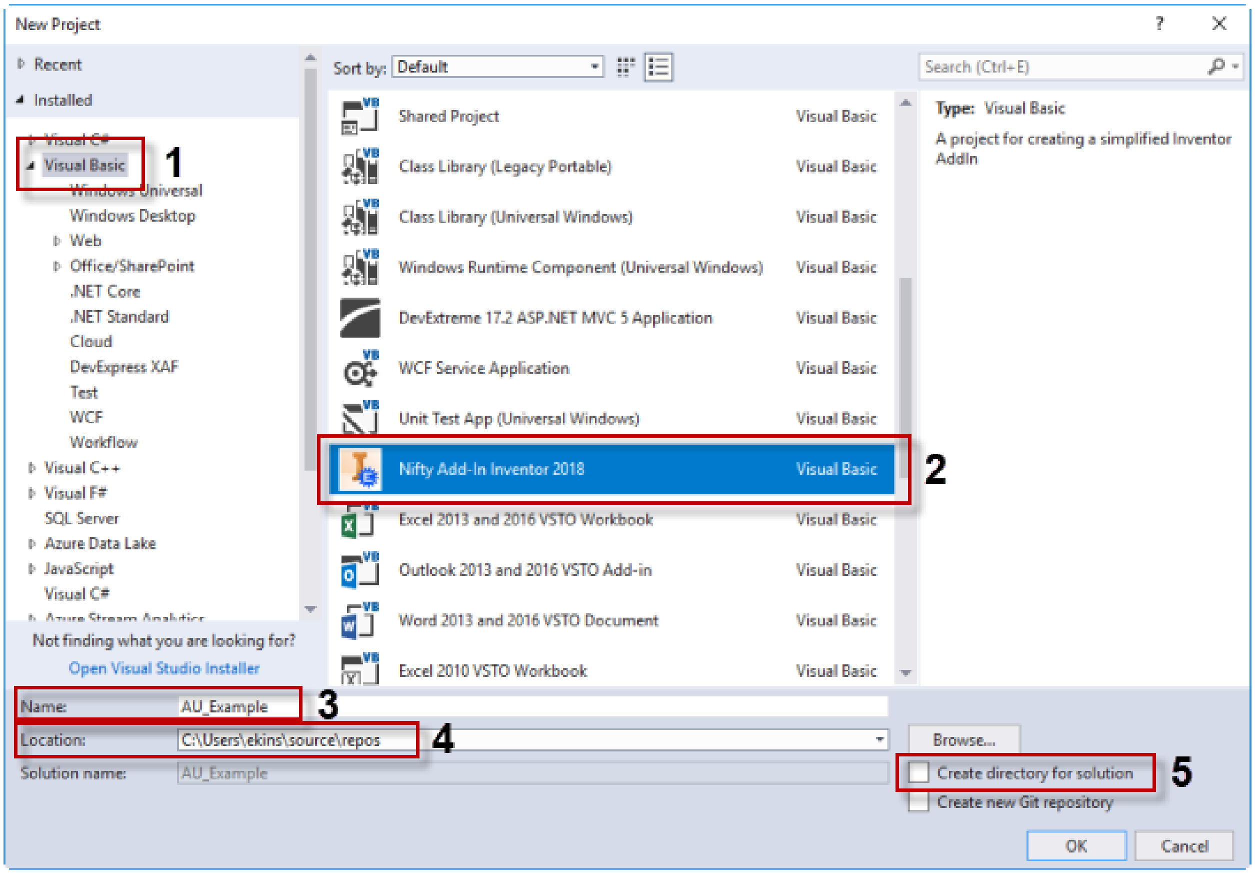
I've developed my own add-in template that I believe makes developing an add-in easier than when using the standard add-in template. My new add-in template can be accessed on my website at https://EkinsSolutions/apps. The rest of this discussion assumes you're using the Nifty Add-In Template.

Creating your Add-In

To create an add-in, you start Visual Studio and run the **New Project** command from the **File** menu, as shown below.



In the **New Project** dialog, choose “Visual Basic” from the “Installed” category on the left and then choose the “Nifty Add-In Inventor 2018” template. Finally, enter the name (without any spaces) and the location where the add-in will be created. I’m using “AU_Example” and it can be located anywhere. If the “Create directory for solution” checkbox is checked, uncheck it and click **OK**.



New Project

Type: Visual Basic

A project for creating a simplified Inventor AddIn

Name: AU_Example

Location: C:\Users\ekins\source\repos

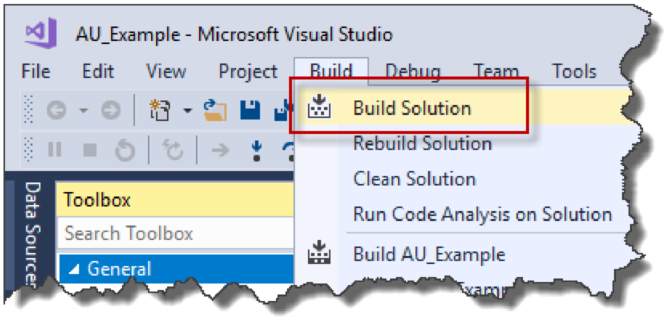
Solution name: AU_Example

□ Create directory for solution

□ Create new Git repository

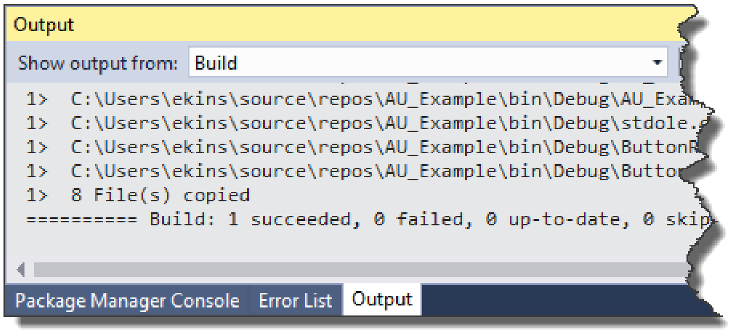
?
Recent
Installed
Visual C#
Visual Basic
1
Windows Universal
Windows Desktop
Web
Office/SharePoint
.NET Core
.NET Standard
Cloud
DevExpress XAF
Test
WCF
Workflow
Visual C++
Visual F#
SQL Server
Azure Data Lake
JavaScript
Visual C#
Azure Stream Analytics
Not finding what you are looking for?
Open Visual Studio Installer
Sort by: Default
Search (Ctrl+E)
Shared Project Visual Basic
Class Library (Legacy Portable) Visual Basic
Class Library (Universal Windows) Visual Basic
Windows Runtime Component (Universal Windows) Visual Basic
DevExtreme 17.2 ASP.NET MVC 5 Application Visual Basic
WCF Service Application Visual Basic
Unit Test App (Universal Windows) Visual Basic
Nifty Add-In Inventor 2018 Visual Basic
2
Excel 2013 and 2016 VSTO Workbook Visual Basic
Outlook 2013 and 2016 VSTO Add-in Visual Basic
Word 2013 and 2016 VSTO Document Visual Basic
Excel 2010 VSTO Workbook Visual Basic
3
4
Browse...
5
OK
Cancel

Now, you can build the project to create the DLL for your add-in by running the **Build Solution** command in the **Build** menu as shown below.



AU_Example - Microsoft Visual Studio

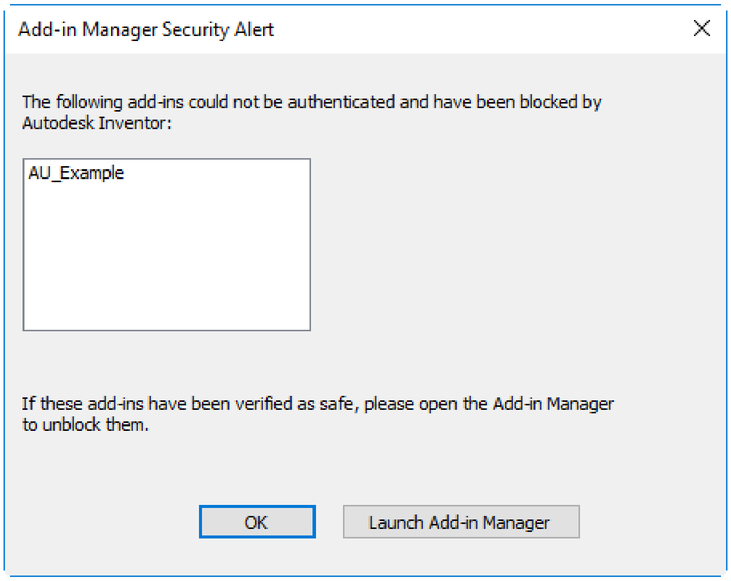
In the “Output” tab at the bottom of Visual Studio, you should see that the build was successful. If you don’t see the output window, press Ctrl+Alt+O.



Show output from: Build

```
1>  C:\Users\ekins\source\repos\AU_Example\bin\Debug\AU_Exa
1>  C:\Users\ekins\source\repos\AU_Example\bin\Debug\stdole.
1>  C:\Users\ekins\source\repos\AU_Example\bin\Debug\ButtonR
1>  C:\Users\ekins\source\repos\AU_Example\bin\Debug\Butto
1>  8 File(s) copied
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skip
```

Congratulations, you’ve just created your first add-in. Start Inventor to see what happens. If Inventor is already running, shut it down and start it up again. Inventor has implemented some security measures to block any unknown add-ins from running. You’ll need to give explicit permission to allow your add-in to run. When you see the dialog below appear, click **OK** and let Inventor finish starting up.

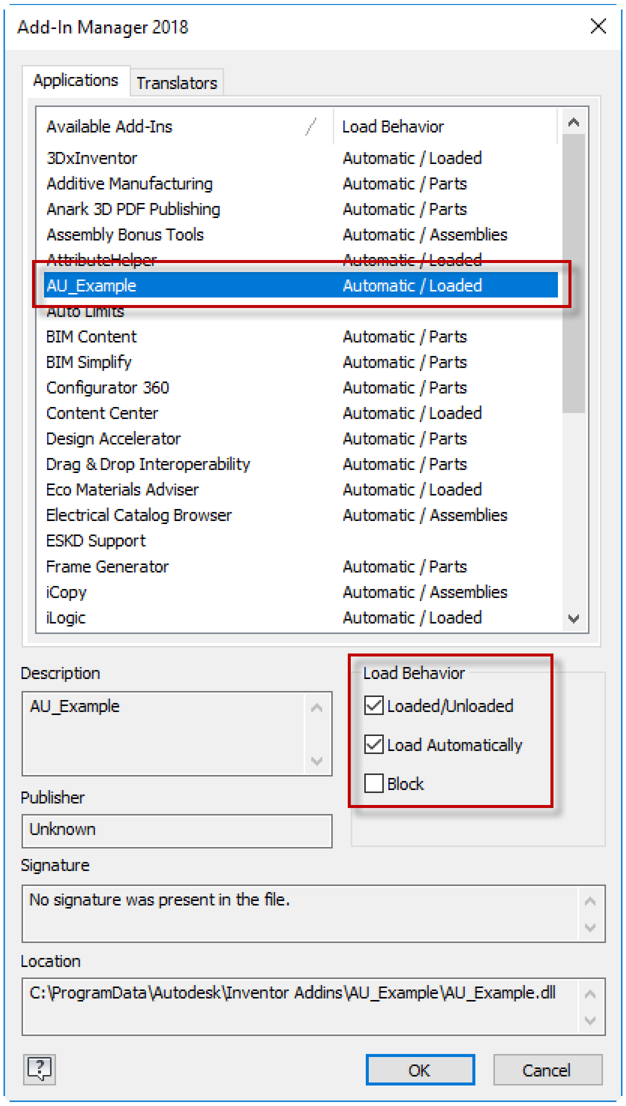


The following add-ins could not be authenticated and have been blocked by Autodesk Inventor:

AU_Example

If these add-ins have been verified as safe, please open the Add-in Manager to unblock them.

Once Inventor is up, run the **Add-Ins** command from the **Tools** tab in the ribbon. Find your add-in in the list, select it, and then change the “Load Behavior” settings so “Loaded/Unloaded” and “Load Automatically” settings are checked, and “Block” is unchecked, as shown below. These settings are remembered by Inventor, so you only need to do this once for a new add-in.



Add-In Manager 2018

Available Add-Ins	Load Behavior
3DxInventor	Automatic / Loaded
Additive Manufacturing	Automatic / Parts
Anark 3D PDF Publishing	Automatic / Parts
Assembly Bonus Tools	Automatic / Assemblies
AttributeHelper	Automatic / Loaded
AU_Example	Automatic / Loaded
Auto Limits	
BIM Content	Automatic / Parts
BIM Simplify	Automatic / Parts
Configurator 360	Automatic / Parts
Content Center	Automatic / Loaded
Design Accelerator	Automatic / Parts
Drag & Drop Interoperability	Automatic / Parts
Eco Materials Adviser	Automatic / Loaded
Electrical Catalog Browser	Automatic / Assemblies
ESKD Support	
Frame Generator	Automatic / Parts
iCopy	Automatic / Assemblies
iLogic	Automatic / Loaded

Description: AU_Example

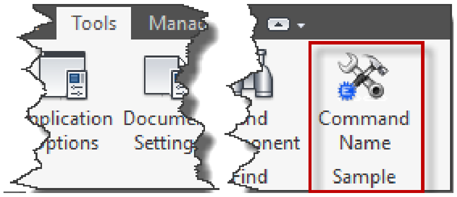
Load Behavior
☑ Loaded/Unloaded
☑ Load Automatically
☐ Block

Publisher: Unknown

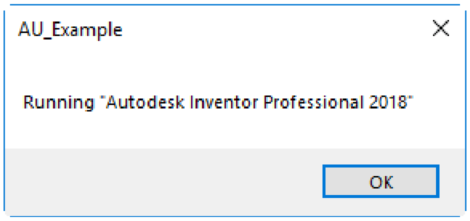
Signature: No signature was present in the file.

Location: C:\ProgramData\Autodesk\Inventor Addins\AU_Example\AU_Example.dll

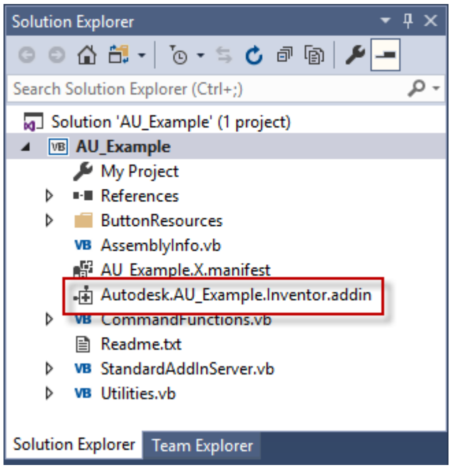
Because you checked the “Loaded” checkbox, your add-in should have loaded when you clicked “OK”. You can open the **Add-In Manager** to verify that its load behavior is now “Automatic / Loaded”. Or the best way to verify that the add-in’s user interface was successfully created is to make sure it is available in Inventor. The add-in template creates a new command and adds a button into the **Tools** tab of the Part ribbon, as shown to the right. Open any part and verify that it exists in the **Tools** tab.



Running the command displays the message box below.



Running "Autodesk Inventor Professional 2018"



Editing the Appearance of Your Add-In

The add-in template uses some defaults to define the various settings of your add-in. You can now change the settings that control how the user sees your add-in. A requirement of all add-ins is to have a .addin file. You'll see it in the "Solutions Explorer" window in Visual Studio as shown to the right. Click on it and it will open a window in Visual Studio to allow you to edit the file. The .addin file is an XML formatted file and its contents are shown below.

```
<Addin Type="Standard">
  <!--Created for Autodesk Inventor 2018 (Version 22.0)-->
  <ClassId>{c322e19b-5e98-46c5-bfe1-fe8cd2e4c173}</ClassId>
  <ClientId>{c322e19b-5e98-46c5-bfe1-fe8cd2e4c173}</ClientId>
  <DisplayName>AU_Example</DisplayName>
  <Description>AU_Example</Description>
  <Assembly>AU_Example.dll</Assembly>
  <LoadOnStartUp>1</LoadOnStartUp>
  <UserUnloadable>1</UserUnloadable>
  <Hidden>0</Hidden>
  <SupportedSoftwareVersionGreaterThan>21..</SupportedSoftwareVersionGreaterThan>
  <DataVersion>1</DataVersion>
  <UserInterfaceVersion>1</UserInterfaceVersion>
</Addin>
```

The *ClassId* and *ClientId* elements are the unique identifiers of your add-in. By default, they are the same as each other and there's no reason to change them. Each time you create a new add-in project a new ID is generated so each add-in is unique.

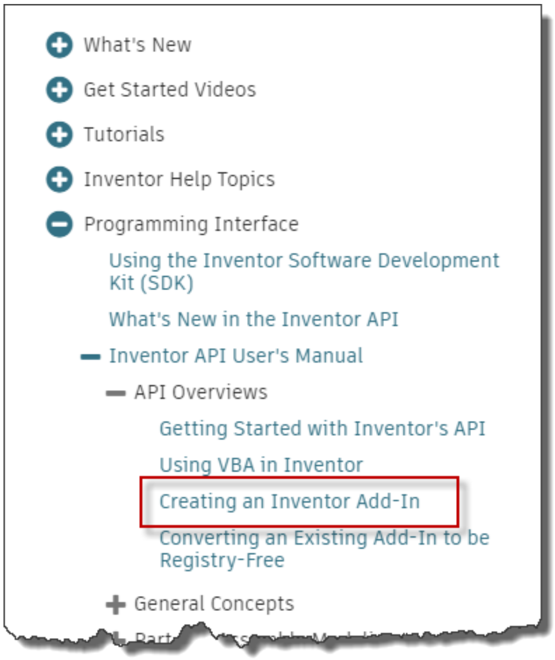
The *DisplayName* element is the name of the add-in as it is displayed in the Add-In Manager and the *Description* element is the description seen in the Add-In Manager. You'll probably want to edit these so they're more user-friendly than the default project name. This is where you can change the name to include spaces. I'm changing the name to "AU Example Add-In" and the description to "This is the example add-in for the AU class on Inventor add-ins."

The *Assembly* element is the file name of the add-in DLL. By default, it is the file name without any path. You can specify either the filename using either a full or relative path. My suggestion is to leave it as-is with just the name of the file. Inventor will find it because it looks for the add-in

DLL using a relative path with respect to the location of the .addin file. In this case, since the DLL is in the same directory as the .addin file, Inventor finds it.

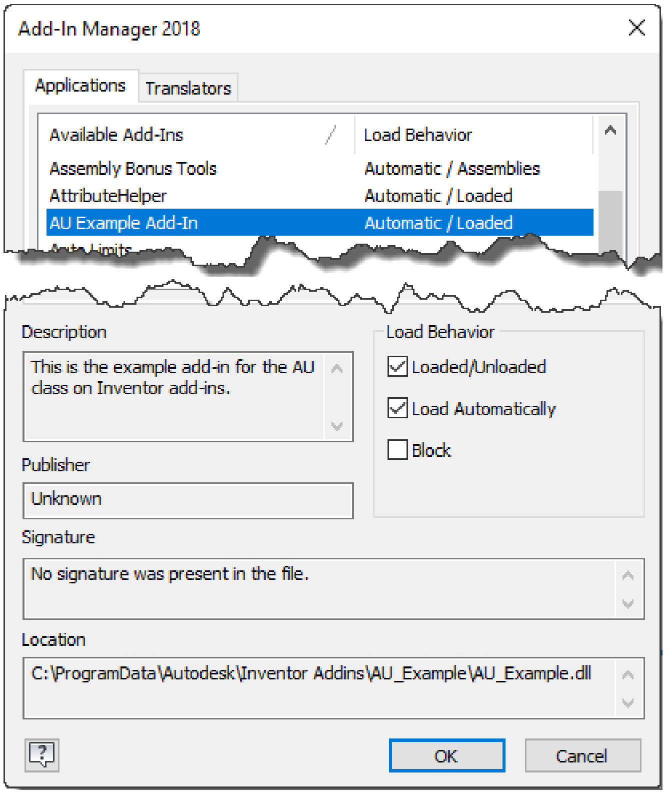
The remaining element of interest is the “SupportedSoftwareVersion” element. It defines when an add-in should be loaded. This element specifies which version(s) of Inventor the add-in is valid for. In this example, it is being loaded by Inventor 2018 and later. Inventor 2018 is version 22 and this is specifying that the add-in should load for versions later than 21.

You can read more about other version options and some other settings that can be defined in the .addin file in the Inventor API help topic shown below.



- What's New
- Get Started Videos
- Tutorials
- Inventor Help Topics
- Programming Interface
  - Using the Inventor Software Development Kit (SDK)
  - What's New in the Inventor API
  - Inventor API User's Manual
    - API Overviews
      - Getting Started with Inventor's API
      - Using VBA in Inventor
      - Creating an Inventor Add-In
      - Converting an Existing Add-In to be Registry-Free
    - General Concepts

After rebuilding the project and restarting Inventor, you can see that the add-in now appears in the Add-In Manager. The name in the list and the description has changed because of editing the .addin file. The “Publisher” and “Signature” fields in the dialog are outside the scope of this paper but involve signing the DLL, which is not required.


Add-In Manager 2018

Applications | Translators

Available Add-Ins	Load Behavior
Assembly Bonus Tools	Automatic / Assemblies
AttributeHelper	Automatic / Loaded
AU Example Add-In	Automatic / Loaded
Auto Limits	

Description
This is the example add-in for the AU class on Inventor add-ins.

Load Behavior
☑ Loaded/Unloaded
☑ Load Automatically
☐ Block

Publisher
Unknown

Signature
No signature was present in the file.

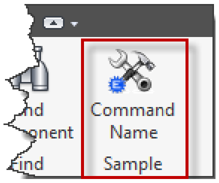
Location
C:\ProgramData\Autodesk\Inventor Addins\AU_Example\AU_Example.dll

Editing the Behavior of Your Add-In

You now have a running add-in that shows up correctly in the Add-In Manager but it doesn’t have the behavior you want. You most likely want the command button to appear in a different location and have a different icon. And you certainly want it to do something different when you click the button. Here are the steps to changing each of these.

Defining the Name and Icon of the Command

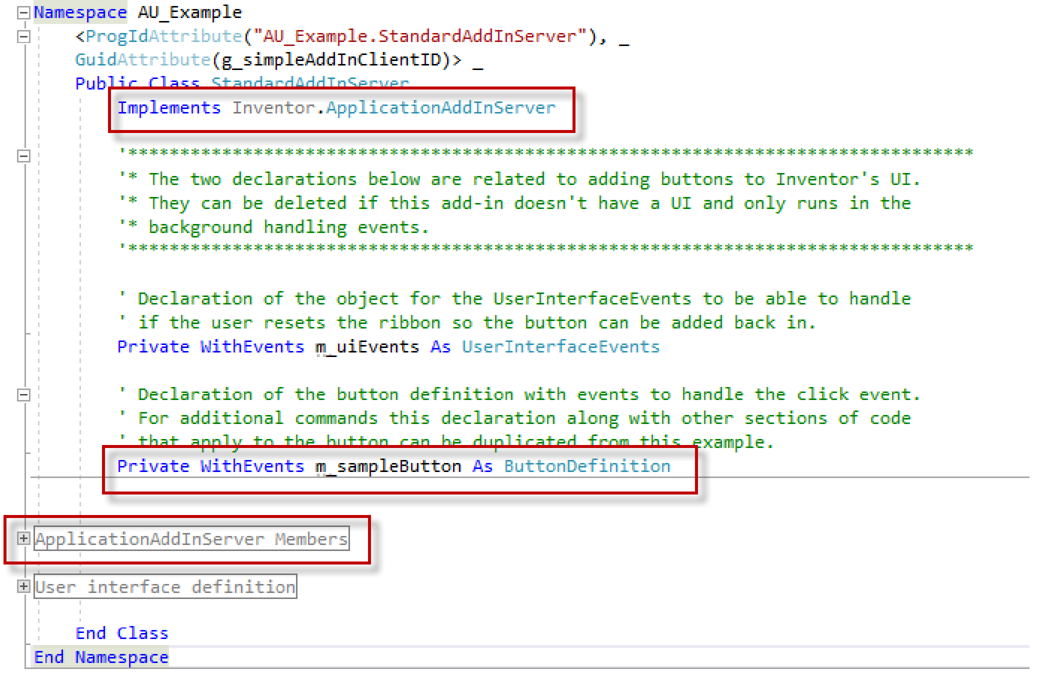
Before looking at the behavior, let’s look at how to change how the command looks in the user interface. First, we’ll look at two things that identify this command; its name and icon. The command’s default name is “Command Name” and the icon is a default icon showing a hammer and wrench. To change the name, you’ll need to edit the code that creates the command.



In the “Solution Explorer” click on “StandardAddInServer.vb” to open that file in Visual Studio. For an add-in to work, it must implement an interface that Inventor has specified. Basically, this means that the add-in must expose an API object and that object must support some specific functions. When Inventor starts, it looks for all available add-ins (how this is done is discussed later) and then loads them. As part of the loading, Inventor calls one of the functions on this defined interface. It's within this function that the add-in uses the Inventor API to create its command and positions it within the user interface.

The code for StandardAddInServer.vb is shown below with three lines highlighted. The `Implements` statement indicates that this class is implementing the Inventor add-in interface.

The next line declares a variable for the add-in's command. An add-in can support many commands. For example, Tube and Pipe has many commands, all defined within the single add-in. The Nifty Add-In template creates an add-in with a single command. You can edit that command to be yours and use it as an example as you add additional commands. The second highlighted line below is the declaration of a variable that is type ButtonDefinition. A ButtonDefinition object defines how a button will be displayed in the user interface and it reacts when the button is clicked. You'll need a line like the one below for each button defined by your add-in. The "WithEvents" keyword specifies that the object can support events, which we need in this case to get an event notification when the button is clicked. The name of this variable can be any valid Visual Basic variable name. The template uses the notation of appending an "m_" to the name to indicate it is a "member variable" in the class. This means that it's available globally within the class. You can also change the name of "m_sampleButton" to a name that better matches what your command does.



```
Namespace AU_Example
    <ProgIdAttribute("AU_Example.StandardAddInServer"), _
    GuidAttribute(g_simpleAddInClientID)> _
    Public Class StandardAddInServer
        Implements Inventor.ApplicationAddInServer

        '**********************************************************************************
        '* The two declarations below are related to adding buttons to Inventor's UI.
        '* They can be deleted if this add-in doesn't have a UI and only runs in the
        '* background handling events.
        '**********************************************************************************

        ' Declaration of the object for the UserInterfaceEvents to be able to handle
        ' if the user resets the ribbon so the button can be added back in.
        Private WithEvents m_uiEvents As UserInterfaceEvents

        ' Declaration of the button definition with events to handle the click event.
        ' For additional commands this declaration along with other sections of code
        ' that apply to the button can be duplicated from this example.
        Private WithEvents m_sampleButton As ButtonDefinition

ApplicationAddInServer Members

User interface definition

    End Class
End Namespace
```

The last highlighted line in the code above is the "ApplicationAddInServer Members" region. This is a block of code that's currently collapsed. Visual Studio supports creating user-defined regions to let you organize your code by collapsing and expanding it to keep your current view uncluttered. Click on the "+" sign to the left of the name to expand the region.

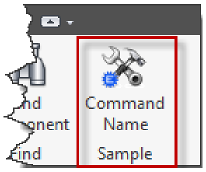
Inside this region is the definition of the Activate, Deactivate, Automation, and ExecuteCommand methods. These are the functions that must be implemented as part of implementing the ApplicationAddInServer interface. The Activate and Deactivate are the most important. The Activate Sub is called by Inventor when the add-in is loaded and the Deactivate is called when the add-in is unloaded.

```
 1  Public Sub Activate(ByVal addInSiteObject As Inventor.ApplicationAddInSite,
 2                ByVal firstTime As Boolean) Implements Inventor.ApplicationAddInServer.Activate
 3      Try
 4          ' Initialize AddIn members.
 5          g_inventorApplication = addInSiteObject.Application
 6  
 7          ' Connect to the user-interface events to handle a ribbon reset.
 8          m_uiEvents = g_inventorApplication.UserInterfaceManager.UserInterfaceEvents
 9  
10          '*******************************************************************************
11          '* The remaining code in this Sub is all for adding the add-in into Inventor's UI.
12          '* It can be deleted if this add-in doesn't have a UI and only runs in the
13          '* background handling events.
14          '*******************************************************************************
15  
16          ' Create the button definition using the CreateButtonDefinition function to simplify
17          ' this step.
18          m_sampleButton = Utilities.CreateButtonDefinition("Command" & vbCr & "Name", _
19                                                            "niftyCommandID", _
20                                                            "", _
21                                                            "ButtonResources\SampleButton")
22  
23          ' Add to the user interface, if it's the first time.
24          If firstTime Then
25              AddToUserInterface()
26          End If
27      Catch ex As Exception
28          MsgBox("Unexpected failure in the activation of the add-in ""AU_Example""" & _
29                 vbCrLf & vbCrLf & ex.Message)
30      End Try
31  End Sub
```

On line 1 you can see that one of the arguments being passed into the add-in is called addInSiteObject and is of type ApplicationAddInSite object. This object doesn't do much, but it is used on line 5 to get the Inventor Application object and assign it to a variable. Because of this, the add-in now has access to the entire Inventor API.

Line 18 calls the function that creates the ButtonDefinition object. The CreateButtonDefinition function isn't part of the Inventor API but is a new function I wrote that's defined within the add-in template. It makes it easier to create a ButtonDefinition and has the four arguments as listed below.

DisplayName – The first argument is the display name. This is the name of the command that is displayed on the button. In the example above, it is defined as "Command Name" with a carriage return between them to force it to break into two lines, as shown to the right.



InternalName – The second argument is the internal name. This is a name that must be unique with respect to all other commands. The user never sees this but it serves as the unique ID for this command. There aren't any rules for this name other than it has to be unique. It's recommended to append your name or company name to the name to help guarantee uniqueness. In the command created by the add-in template, it is "niftyCommandID" but you should rename it to something else

Tooltip – The third argument is optional and is the tooltip. If you don't provide this argument or pass in the default value of an empty string, then the display name is used as the tooltip. The tooltip is displayed when the user hovers the mouse over the command button.

IconFolder – The fourth argument defines the icon to use for the command. In the past, setting up the icon has been one of the more difficult parts of creating an add-in but the CreateButtonDefinition function attempts to simplify this. All you need to define the icon is to specify a folder for this argument. That folder should contain two files; 16x16.png and 32x32.png. These are the images that will be displayed on the button. Inventor can display large and small buttons, so it needs two images; 16x16 pixels and 32x32 pixels.

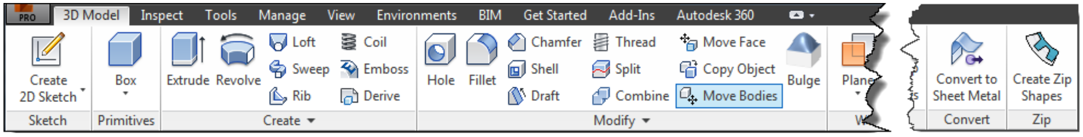
In the example created by the add-in template, the folder is "ButtonResources\SampleButton". This is a path that is relative to the add-in DLL. For each new command you add, you should create a new folder within the ButtonResources folder. That new folder will contain the 16x16.png and 32x32.png file for that command. If you use Windows Explorer to browse the contents of your add-in project, you'll see that there is already a ButtonResources folder and in that folder is the SampleButton folder that contains the two png files.

Positioning Your Button

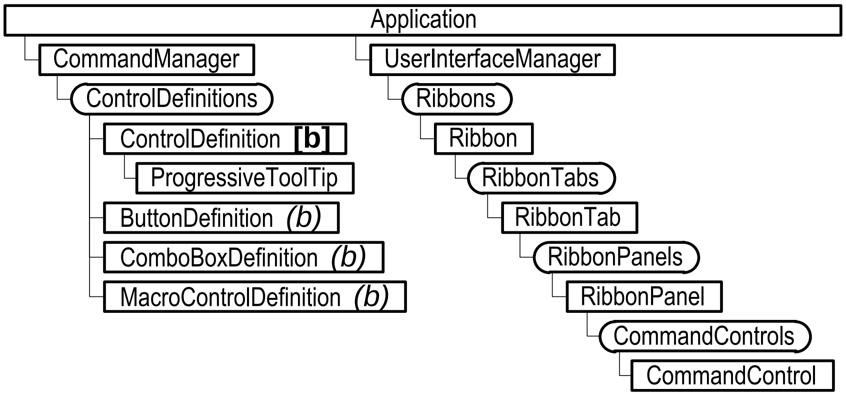
The code above creates a ButtonDefinition, which defines what the button will look like but it doesn't define where the button will exist in the user interface. You do this by creating a new command control. A command control defines the location in the user interface and is what the user sees and clicks. The command control references the button definition to get the information it needs to display itself. Because of this, you can create a single ButtonDefinition and then create multiple controls that reference it. For example, if I have a generic command, like a calculator, I might want to add it to all of the ribbons. Even though there are multiple buttons, they all end up triggering the same event when any of them are clicked because it's the single button definition that handles the event. Also, if I edit the button definition, for example, make it disabled, all of the controls referencing that button definition will become disabled.

You have complete control over where to position your button, but because of that flexibility, it's not particularly simple to do. To position a button you need to have an understanding of the ribbon and the associated API.

The ribbon provides a hierarchical structure to access Inventor's commands. At the top of the hierarchy is the *ribbon*. The part ribbon is shown below as an example. In Inventor there are 7 ribbons. The most commonly used ribbons are ZeroDoc (when no documents are open), Part, Assembly, Drawing, and Presentation. Within each ribbon are a series of *tabs*. In the Part ribbon, as shown below, there are the tabs "3D Model", "Inspect", "Tools", "Manage", "View", etc. Each tab contains *panels*. The "3D Model" tab contains the panels "Sketch", "Primitives", "Create", "Modify", etc. And finally, each tab contains *controls*, which are the individual buttons.



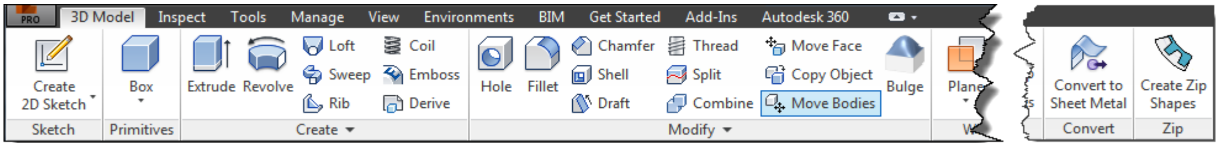
Below is the API object model for the ribbon related objects that you use to create a button and add it to the ribbon.



Below is the code created by the add-in template that adds the control for the sample command. It traverses the Ribbon hierarchy to get a specific ribbon, tab, and panel and then creates a new CommandControl. On line 8 it gets the ribbon named “Part”. On line 11 it gets the tab in the part ribbon named “id_TabTools” which is the “Tools” tab. On lines 13-18 it attempts to get a panel named “MySample” from the tool tab. On line 20 it checks to see if it was able to successfully get the “MySample” panel and if it didn’t it creates it on line 22. The variable g_addInClientID on line 22 is a global variable the template created that contains the unique ID for the add-in. And finally, on line 26 it uses the AddButton method to create a new CommandControl in the panel, passing in the ButtonDefinition that was previously created.

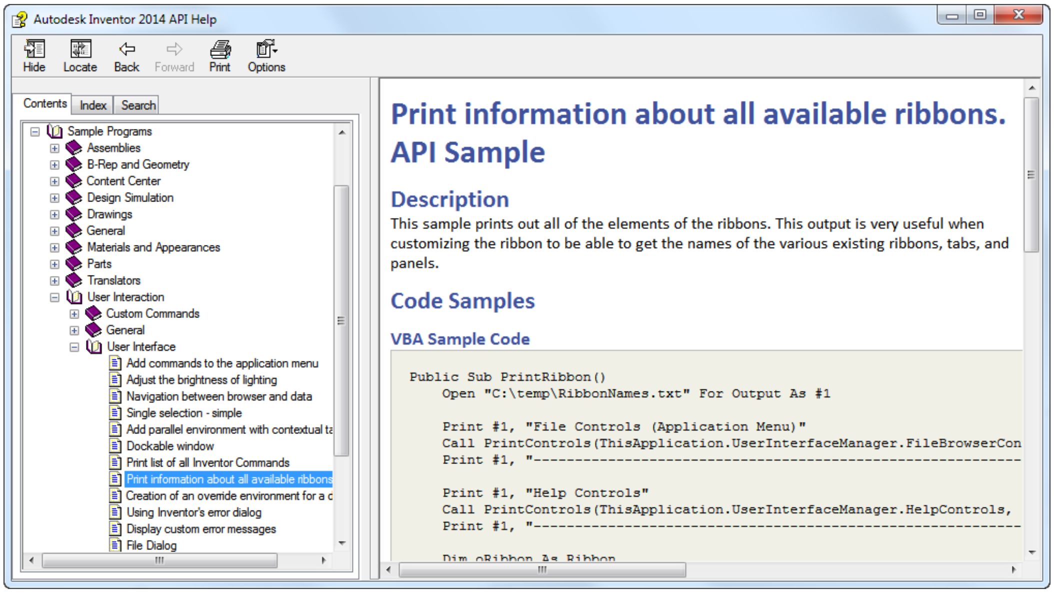
```
1  ' Adds whatever is needed by this add-in to the user-interface. This is
2  ' called when the add-in loaded and also if the user interface is reset.
3  Private Sub AddToUserInterface()
4      ' This sample code illustrates creating a button on a new panel of the Tools tab of
5      ' the Part ribbon. You'll need to change this to create the UI that your add-in needs.
6  
7      ' Get the part ribbon.
8      Dim partRibbon As Ribbon = g_inventorApplication.UserInterfaceManager.Ribbons.Item("Part")
9  
10     ' Get the "Tools" tab.
11     Dim toolsTab As RibbonTab = partRibbon.RibbonTabs.Item("id_TabTools")
12 
13     ' Check to see if the "MySample" panel already exists and create it if it doesn't.
14     Dim customPanel As RibbonPanel = Nothing
15     Try
16         customPanel = toolsTab.RibbonPanels.Item("MySample")
17     Catch ex As Exception
18     End Try
19 
20     If customPanel Is Nothing Then
21         ' Create a new panel.
22         customPanel = toolsTab.RibbonPanels.Add("Sample", "MySample", g_addInClientID)
23     End If
24 
25     ' Add a button.
26     customPanel.CommandControls.AddButton(m_sampleButton, True)
27 End Sub
```

Here's another example to illustrate the concepts. Through the API you can add buttons to existing panels and you can create your own tabs and panels. The example below shows a "Bulge" custom command added to the "Modify" panel and a custom panel called "Zip" where the custom command "Create Zip Shapes" has been added.



The first step in adding a new command is to decide where your button needs to go. Does it have similar functionality to an existing Inventor command so you might want to position it close to that command? Or does it do something entirely different than anything else and it should go on its own panel or even its own tab combined with more of your custom commands? Once you know where you want it to go you need to get the names of the associated ribbon elements. For example, for the "Bulge" command shown above, I needed to get the "Part" ribbon, the "3D Model" tab, and the "Modify" panel. You can even go further and get a specific control within the panel if you want to position your button next to it. By default, it will go to the end of the commands in the panel.

To get each of these ribbon elements you need to know their names. To find out their names you'll want to copy and paste the sample macro from the API help called "Print information about all available ribbons", as shown below. Running the macro will result in the creation of C:\temp\RibbonNames.txt which contains the names of all the ribbon elements.


Autodesk Inventor 2014 API Help
Hide
Locate
Back
Forward
Print
Options
Contents
Index
Search
Sample Programs
Assemblies
B-Rep and Geometry
Content Center
Design Simulation
Drawings
General
Materials and Appearances
Parts
Translators
User Interaction
Custom Commands
General
User Interface
Add commands to the application menu
Adjust the brightness of lighting
Navigation between browser and data
Single selection - simple
Add parallel environment with contextual ta
Dockable window
Print list of all Inventor Commands
Print information about all available ribbons
Creation of an override environment for a c
Using Inventor's error dialog
Display custom error messages
File Dialog

Print information about all available ribbons. API Sample

Description

This sample prints out all of the elements of the ribbons. This output is very useful when customizing the ribbon to be able to get the names of the various existing ribbons, tabs, and panels.

Code Samples

VBA Sample Code

```
Public Sub PrintRibbon()
    Open "C:\temp\RibbonNames.txt" For Output As #1

    Print #1, "File Controls (Application Menu)"
    Call PrintControls(ThisApplication.UserInterfaceManager.FileBrowserCon
    Print #1, "------------------------------------------------------------

    Print #1, "Help Controls"
    Call PrintControls(ThisApplication.UserInterfaceManager.HelpControls,
    Print #1, "------------------------------------------------------------

    Dim oRibbon As Ribbon
```

Using the information in the RibbonNames.txt file I can see that the part ribbon is named “Part”, the “3D Model” tab is named “id_TabModel”, and the “Modify” panel is “id_PanelP_ModelModify”. Using that information, the code below adds a new button to that panel.

```
' Get the part ribbon.
Dim partRibbon As Inventor.Ribbon =
g_inventorApplication.UserInterfaceManager.Ribbons.Item("Part")

' Get the "Model" tab from the part ribbon.
Dim modelTab As Inventor.RibbonTab = partRibbon.RibbonTabs.Item("id_TabModel")

' Get the "Modify" panel from the model tab.
Dim modifyPanel As Inventor.RibbonPanel = modelTab.RibbonPanels.Item("id_PanelP_ModelModify")

' Add the button to the modify panel.
modifyPanel.CommandControls.AddButton(m_bulgeButton, True)
```

Reacting to a Button Click

As described above, the creation of a button involves creating a ButtonDefinition object to define what a button looks like and then the creation of a control to define the button’s location in the ribbon. When the variable for the ButtonDefinition was defined, the “WithEvents” keyword was used to indicate to Visual Basic that you want to handle events that are associated with that object. A ButtonDefinition object supports the OnExecute event. The OnExecute event is fired whenever any of the buttons that reference the ButtonDefinition are clicked. It’s up to you to handle that event and then do whatever the command is supposed to do.

Here’s a brief primer on setting up a ButtonDefinition and handling the OnExecute event. This same process is used when handling any other event in Inventor.

Define the Variable for the Object that Supports Events

This is done by declaring the variable at a place where it will remain in scope for the desired time. For example, the variable for the ButtonDefinition is declared as a member variable of the add-in class so it will remain available for the lifetime of the add-in. Events associated with forms are typically declared within the Form class so they remain available for the lifetime of the form. You declare a variable that will support events by using the “WithEvents” keyword, as shown below.

```
Private WithEvents m_sampleButton As ButtonDefinition
```

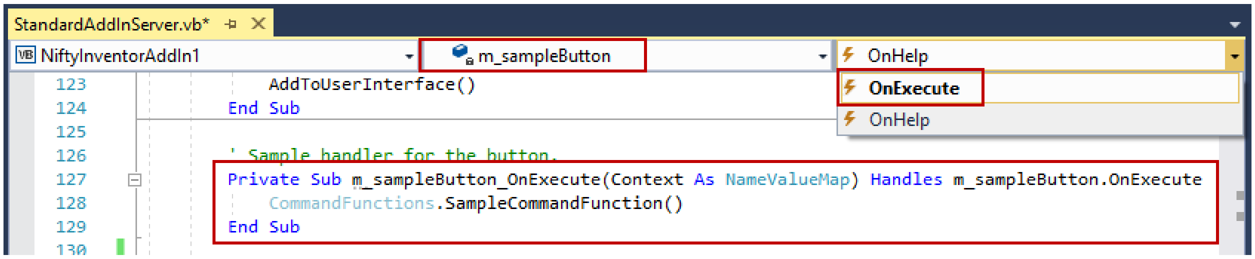
The line above is declaring the variable but it’s not assigning anything to it. That’s done later in the program within the Activate method when the add-in is loaded. In the Nifty Add-In template, you’ll see that it’s declaring another variable using WithEvents that is of type UserInterfaceEvents to be able to listen for a user interface related event. You use this same approach to listen to any of Inventor’s events.

Assigning the Event Variable

Once you declared the variable you need to assign something to it. In the Nifty Add-In template, it calls the Utilities.CreateButtonDefinition method which creates and returns a ButtonDefition object and this is assigned to the `m_sampleButton` variable.

Creating an Event Handler

You declare the variable, assign it, then you need a Sub that acts as the event handler. This is a sub that Inventor calls when that event occurs. Visual Studio makes it easy to create this sub. At the top of the code window are three drop-downs. If I expand the middle one, I see a list of the variables I've declared that support events. In the example below, I've selected "m_sampleButton". Once a variable has been selected from the list, the drop-down on the right will display a list of the events that object supports. In this case, I've selected the OnExecute event and Visual Studio automatically created a new Sub called m_sampleButton_OnExecute that handles the OnExecute event. Now, when the button is clicked this Sub will be called by Inventor. You can do anything you want inside this sub. In this example, it calls another function that does the work of the command.



```
            AddToUserInterface()
        End Sub

        ' Sample handler for the button.
        Private Sub m_sampleButton_OnExecute(Context As NameValueMap) Handles m_sampleButton.OnExecute
            CommandFunctions.SampleCommandFunction()
        End Sub
```

Writing the Command Code

Everything I've talked about so far is about the available programming tools and the creation of the basic structure of the add-in. Once the user clicks the button, then it's all about what you do in response to that click. The code that you write to respond to the click is add-in independent and could have been written as an iLogic rule that only uses the Inventor API or an EXE Macro so talking about the code is a somewhat generic topic since it doesn't have anything to do with the specifics of an add-in. However, because I believe Visual Studio is a much better environment for writing code I think for your programs that don't need any iLogic specific functionality it's best to write either EXE's or add-ins.

It's likely that you already have some code that could be used in an add-in or EXE Macro. Your existing code might have been written in VBA or iLogic. Most of the sample code in the Inventor API help is VBA code and variations of the samples might be useful in your programs. Here's some information to help you in converting existing code.

Converting iLogic Programs

First, what is iLogic? It's essentially another IDE (Integrated Development Environment) for Inventor. It supports the Visual Basic .NET language and provides access to the Inventor API. It also provides its own library with some additional functions you can use in your programs that attempt to simplify some operations that the iLogic designers thought would be common when configuring parts. For part configuration, iLogic is a great tool but as a general programming environment, it leaves a lot to be desired. On the Inventor Customization forum, most of the questions that I see regarding iLogic aren't iLogic questions at all but are general Inventor API questions and the developer is using the iLogic interface to write their program. In many cases, their problems are easily solved just by copying the code over into a Visual Studio project and letting it automatically find the problems.

Converting an iLogic program to an add-in command or an EXE macro can be as easy as copy and paste. This is true in the case where the program is making Inventor API calls and not calling any of the iLogic specific functions. If it is calling iLogic specific functionality then you'll need to determine the extent of their use and find alternatives using standard Inventor API calls or other API's. However, because iLogic is using VB.NET, just the same as Visual Studio, the general code is the same and calls to the Inventor API will be the same.

Converting VBA Programs

Even though VBA and Visual Studio both use the Visual Basic language it's best to think of them as using two different languages. VBA is based on Microsoft's original Visual Basic which the latest version is Visual Basic 6, and was released in 1998. Visual Studio uses Visual Basic .NET which is based on .NET technology launched in 2002 and it continues to be updated. In 2005, the ".NET" portion of the name was dropped so it's now called "Visual Basic".

If you want to use an existing VBA program in an add-in or EXE Macro, it means converting the code from the old Visual Basic 6 language into the new Visual Basic .NET language. Even though there are differences in the languages they are more similar than they are different so it's usually quite easy to copy and paste VBA code and then make a few changes to update it to Visual Basic .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 and catch any of these potential problems. The issues I think you're most likely to encounter are listed below.

1. **The Set statement is no longer needed.** In VBA when you assign an object to a variable you must use the `Set` statement. This is no longer required in Visual Basic and in fact, is not allowed. You'll get an error for any `Set` statement, but to fix it all you need to do is delete `Set` from the line.

2. **Forms.** Although VBA supports the creation of forms, they're quite primitive. There isn't a way to convert VBA forms to Visual Basic .NET forms, but you'll likely want to redesign them anyway using the expanded capabilities of Visual Basic .NET and you can still re-use the code behind the form by copying and pasting it into the Visual Basic .NET program. For example, the code you wrote to react to a button click can be reused, but the dialog and the button on the dialog will have to be recreated.

3. **The global variable ThisApplication isn't available in Visual Basic .NET.** It was shown earlier how an add-in gets the Application object through the Activate method. You'll need to somehow make this available to the code you copied from VBA. The Nifty Add-In template gets the Application object and assigns it to a global variable that can be used anywhere else in the program. This just means replacing anywhere "ThisApplication" appears with the global variable "g_inventorApplication".

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

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

4. **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 to create the fully qualified name.

5. **Method arguments default to ByVal.** In VBA, function arguments default to ByRef which means the sub or function can change the values and the modified values are passed back to the caller. With ByVal, the value of the variable is passed in but is local within the sub or function so even if the value changes it's not passed back to the caller. Here's an example to illustrate what this means. The VBA Sub below takes a feature as input and returns suppression and dimension 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. 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. In Visual Basic .NET, if you don't specify ByRef or ByVal, it defaults to ByVal. Because of that, this example won't run correctly because the Suppressed and DimensionCount arguments won't return the correct values. They need to have the ByRef keyword added to the arguments for the sub to work as expected.

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

6. **Arrays have a lower bound of 0 (zero).** 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 Visual Basic .NET statement below creates an array that contains 9 doubles.

```
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 of an array 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 the values in the array down by one index. You'll need to change the index values everywhere the array is used.

7. **Arrays of changing size are handled differently in VB.NET.** This, and the next item are a couple of array related issues that you might run into. You can't specify the type when you use the ReDim statement to 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)
```

8. **Declaring an array in VB.NET does not initialize it.** This will likely be another common problem encountered that's not obvious what the problem is from the error. The VBA code below will fail in .NET resulting in a type mismatch error. This is easily fixed by initializing the value to an empty array in the declaration, 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)
```

9. **Some data types are different.** There are two changes here that can cause problems. First, the VBA Long type is equivalent to the VB.NET Integer type. If you're calling an Inventor API 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. Changing the declaration from Long to Integer will 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 equivalent Object type instead.

10. **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 of the variable strSuppressed, (highlighted in the sample below), 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 now 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
```

11. **Events.** The concepts and basic mechanics of how to use events are 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. 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 defined it as 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 change, rather than copying and pasting the entire event handler sub from VBA it's best to create a new event handler in VB.NET letting Visual Studio create the code for you and then copy and paste the contents of the sub from VBA.

12. **Parentheses are now required around property and method arguments.** The use of parentheses is a bit confusing in VBA because of its inconsistent requirements. When you copy and paste the code into VB.NET it will point out missing parentheses as an error that you ca easily fix. The Call statement is still supported but is no longer needed.

Cool stuff in Visual Basic .NET

As we see from the above discussion, there are some differences between VBA and Visual Basic.NET which cause some issues when porting code between them, however, there are a lot of new capabilities in Visual Basic .NET that you'll certainly miss when you try writing VBA code again. Here's a short list of some of my favorites.

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

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

You can even do this in a `For` statement:

```
For i As Integer = 1 To 20
```

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. The .NET libraries provide a much richer set of functionalities for math functions, string handling, file handling, working with XML files, the registry, and most everything else.

4. Incrementing a variable. There's now some simpler syntax for incrementing a variable.

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

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

Creating Forms

You create a dialog as a Visual Basic form and typically display it in response to the user clicking 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 to display it as a modal dialog. Modal means that the user can't do anything else while the dialog is displayed but they must interact and dismiss it before moving on.

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

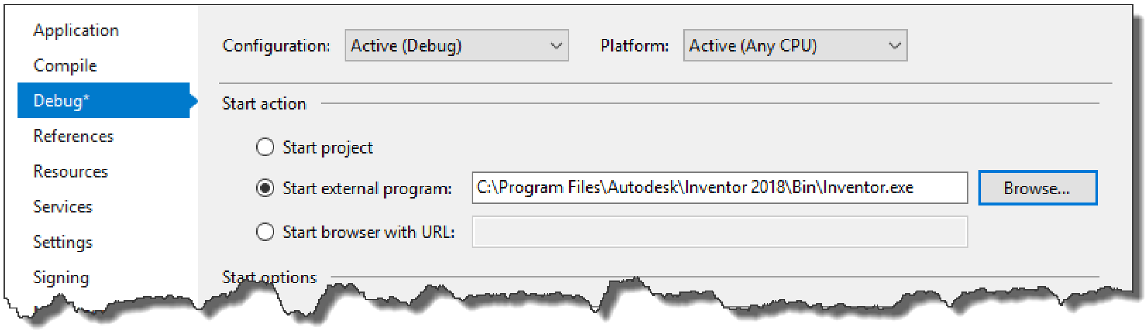
One issue with displaying a dialog is that by default it is independent of the Inventor main window. This can cause a few problems; the Inventor window can cover the dialog, the dialog is still displayed if the Inventor window is minimized, and key presses that represent keyboard shortcuts are stolen by Inventor. To get around these problems you can make the dialog a child of the Inventor window. The sample code above does this by using the WindowWrapper utility class, which is part of an add-in project created using the Nifty Add-In template.

Debugging Your Add-In

So far everything that's been discussed applies to all the varieties of Visual Studio. However, debugging is where there is a big difference. The Express edition of Visual Studio does not support debugging a *class library* type of project, which is what an add-in is, as easily as the other versions of Visual Studio. First, let's cover debugging using the Professional and Community versions of Visual Studio.

Debugging With Visual Studio Community and Professional

Debugging an add-in is an interesting problem because an add-in doesn't run on its own but is loaded and called by another program. The program loading and calling an add-in is Inventor. To debug an add-in, you need Inventor to load it and make the necessary calls while you're monitoring everything in the debugging environment. To set up debugging, Run the "**Properties…**" command, which is the last command in the **Project** menu. This will display the dialog shown below. Select the **Debug** tab and then select the **Start external program** option and browse to locate Inventor.exe as shown below.



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

Start action

○ Start project

● Start external program: C:\Program Files\Autodesk\Inventor 2018\Bin\Inventor.exe

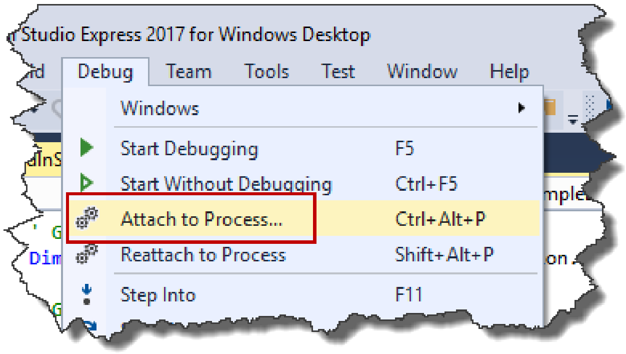
○ Start browser with URL:

To start debugging, run the **Start Debugging** command in the **Debug** menu or press F5. Visual Studio will start Inventor and be in a state that you can debug the add-in. Any breakpoints you've inserted into your program will stop execution and allow you to step through your program. When a breakpoint is hit you can examine values, step through code, and continue running.

Debugging with Visual Studio Express

As I said earlier, debugging an add-in isn't as easy using Visual Studio Express. It doesn't support automatically starting and connecting to an external program. However, you can still debug with the Express version by performing these steps manually.

1. Compile your project.
2. Start Inventor. Make sure that the "Load Automatically" option in the **Add-In Manager** for your add-in is NOT checked. You don't want it to load automatically because you need to be able to control the timing of when it loads.
3. In Visual Studio, run the **Attach to Process…** command in the **Debug** menu, as shown below.

   
   Studio Express 2017 for Windows Desktop
   Debug
   Team
   Tools
   Test
   Window
   Help
   Windows
   Start Debugging F5
   Start Without Debugging Ctrl+F5
   Attach to Process... Ctrl+Alt+P
   Reattach to Process Shift+Alt+P
   Step Into F11

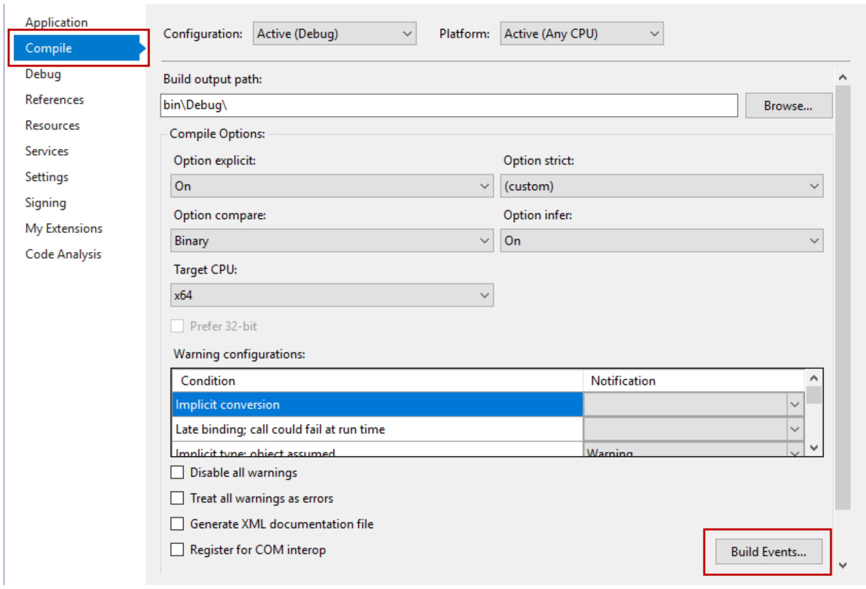
   In the "Attach to Process" dialog, choose "Inventor.exe" from the list and click "Attach". You might notice some small changes in Visual Studio because it's now in a debug mode, but there's not a big change at this point. What's happened is that Visual Studio is running your add-in and has attached itself to Inventor so when Inventor loads the add-in it will load the version of your add-in that is running in Visual Studio.
4. In Visual Studio, add any desired breakpoints to your code.
5. In Inventor, run the Add-Ins command to open the Add-In Manager, select your add-in, check the "Loaded/Unloaded" checkbox and click OK. Inventor will now load your add-in and execution will be paused when any breakpoints are hit.

What Happens When You Compile Your Add-In?

There's something that is happening behind the scenes when you compile your add-in that you should be aware of to better understand what's going on in case you need to make changes, or something isn't working as expected.

When you start Inventor, it looks in a series of folders on your computer for .addin files. It reads the contents of the .addin files it finds and if the add-in meets the criteria to be loaded (for example, it's the correct version), Inventor loads the add-in DLL. Inventor isn't looking in your Visual Studio project folder for .addin files so it won't find your add-in. To make debugging easier, the Nifty Add-In Template has set up the Visual Studio project so your add-in files are being copied to one of the folders where Inventor looks for .addin files. This happens automatically every time you compile your add-in so the files where Inventor is looking are always up to date.

This is accomplished by using a "Build Event" in Visual Studio. In the properties dialog for the project in the "Compile" tab you'll see a "Build Events…" button, as shown below.



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

Build output path: bin\Debug\

Compile Options:

Option explicit: On   Option strict: (custom)

Option compare: Binary   Option infer: On

Target CPU: x64

□ Prefer 32-bit

Warning configurations:

Condition	Notification
Implicit conversion	
Late binding; call could fail at run time	
Implicit type; object assumed	Warning

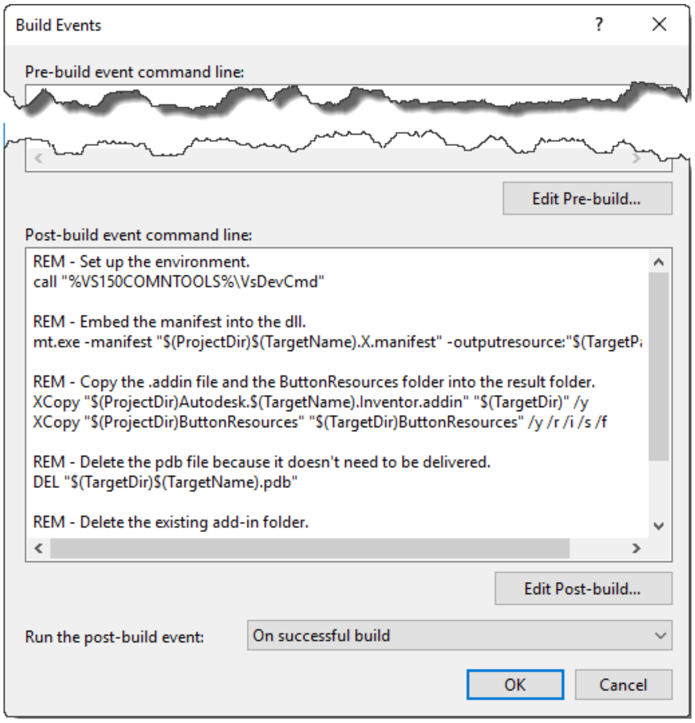
□ Disable all warnings

□ Treat all warnings as errors

□ Generate XML documentation file

□ Register for COM interop

Clicking this brings up the "Build Events" dialog where you can enter DOS commands that will be executed before and after your build. The add-in template sets up some post-build steps, as shown below. These are run after a successful compile.



Pre-build event command line:

Post-build event command line:

```
REM - Set up the environment.
call "%VS150COMNTOOLS%\VsDevCmd"

REM - Embed the manifest into the dll.
mt.exe -manifest "$(ProjectDir)$(TargetName).X.manifest" -outputresource:"$(TargetPa

REM - Copy the .addin file and the ButtonResources folder into the result folder.
XCopy "$(ProjectDir)Autodesk.$(TargetName).Inventor.addin" "$(TargetDir)" /y
XCopy "$(ProjectDir)ButtonResources" "$(TargetDir)ButtonResources" /y /r /i /s /f

REM - Delete the pdb file because it doesn't need to be delivered.
DEL "$(TargetDir)$(TargetName).pdb"

REM - Delete the existing add-in folder.
```

Run the post-build event: On successful build

The ability to define build events is supported by Visual Studio Professional and Community but not Express. Actually, Express doesn't officially support build events because it doesn't provide the user interface to let you define them, but if they're there they do work. So, the build events that were defined by the project template are there and are executed when you build with Express but you don't have the form shown above to make changes to them. These settings are stored in the project's .vbproj file and it's possible to edit them directly using any text editor.

Deploying Your Add-In

Now that you've got a working add-in, how do you deploy it for others to use? There are two approaches to deploying your add-in. The first is the simplest, which is to copy the add-in folder onto the target machine. The second is to create an installer. Both are discussed below.

Method 1 – Copy the Folder

The first and easiest way to deliver an add-in is to copy the add-in folder to the target computer. All you need to do is copy the entire add-in folder to the correct location and it will work. The simple add-in template creates a directory on your machine that contains all the files associated with the add-in (.addin, .DLL, icon files, etc.). You can copy this directory to any other machine and the add-in will be available there. There are several directories where Inventor looks for add-ins, however, I recommend either of the following two:

```
%appdata%\Autodesk\ApplicationPlugins

%programdata%\Autodesk\Inventor Addins
```

The `appdata` directory is user specific so if the computer is used by multiple users it will only be available to the single user where this is copied. Because it is user specific you don't need administrative privileges to copy files it there. This is the folder where a project created using the add-in template is copied by the post-build step.

The `programdata` directory is for all users and you will need administrative privileges.

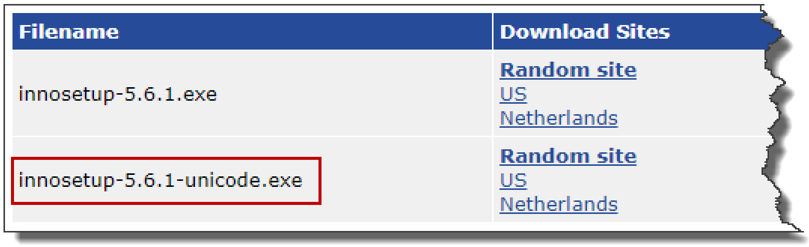
To remove the add-in from the machine you delete the folder.

Method 2 – Create an Installer

The second method takes more work on your side but is simpler for the person installing it because they just run the installer and it puts everything where it should go. The installer copies the needed files to the correct location so it's about as simple as an installer can get. Another advantage with using an installer is that it will show up in the installed programs list and the user can easily uninstall it from there.

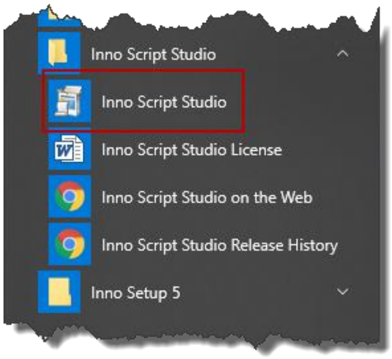
There are many tools to create an installer, some are free and some cost 100's of dollars. For example, Visual Studio Professional and Community can be used to create installers. However, I've found a free tool that I've been happy with and is what I use to create all my Windows installers. It also allows those of you using Visual Studio Express to create an installer. 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 create a setup using Inno Setup.

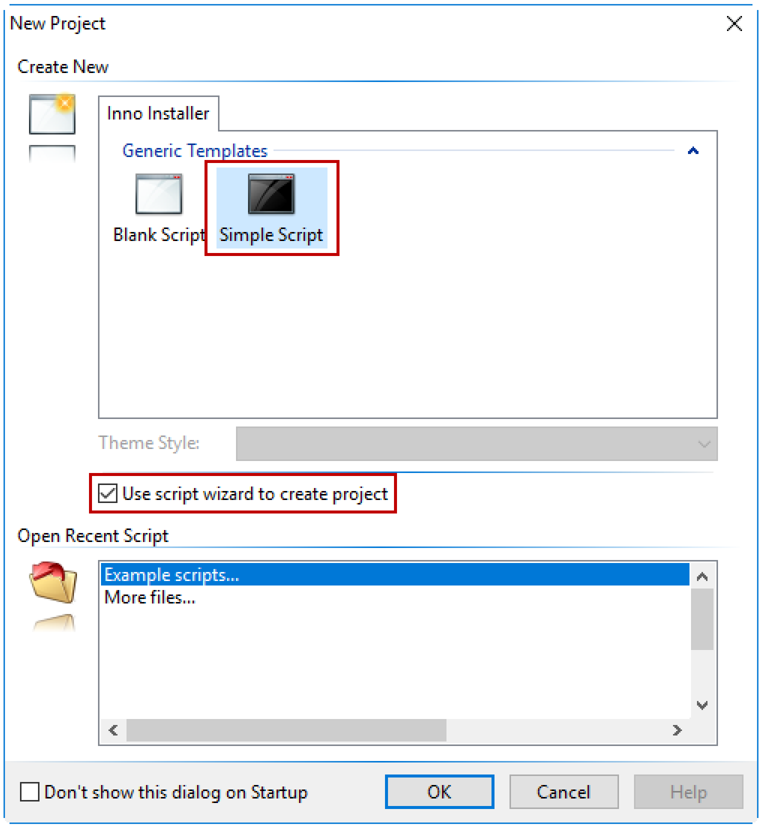
1. Download the Unicode QuickStart pack, as shown below. Install the pack, taking all the defaults. The QuickStart pack includes Inno Setup and some additional utilities to make using it easier.



Filename	Download Sites
innosetup-5.6.1.exe	Random site US Netherlands
innosetup-5.6.1-unicode.exe	Random site US Netherlands

2. When you run **Inno Script Studio** from the start menu you'll be presented with the dialog shown below. Select the "Simple Script" template and make sure the "Use script wizard to create project" is checked.

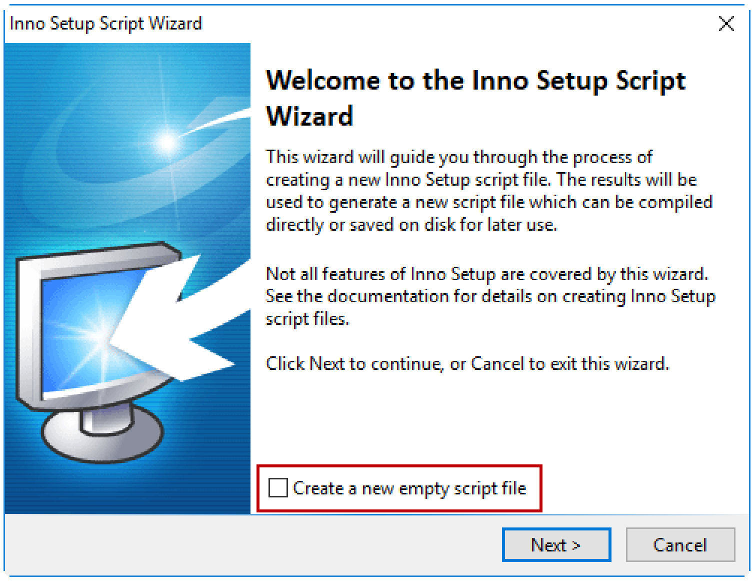

Inno Script Studio
Inno Script Studio
Inno Script Studio License
Inno Script Studio on the Web
Inno Script Studio Release History
Inno Setup 5



☑ Use script wizard to create project

□ Don't show this dialog on Startup

3. On the Welcome screen, make sure “Create a new empty script file” is unchecked and click “Next”.



Inno Setup Script Wizard

Welcome to the Inno Setup Script Wizard

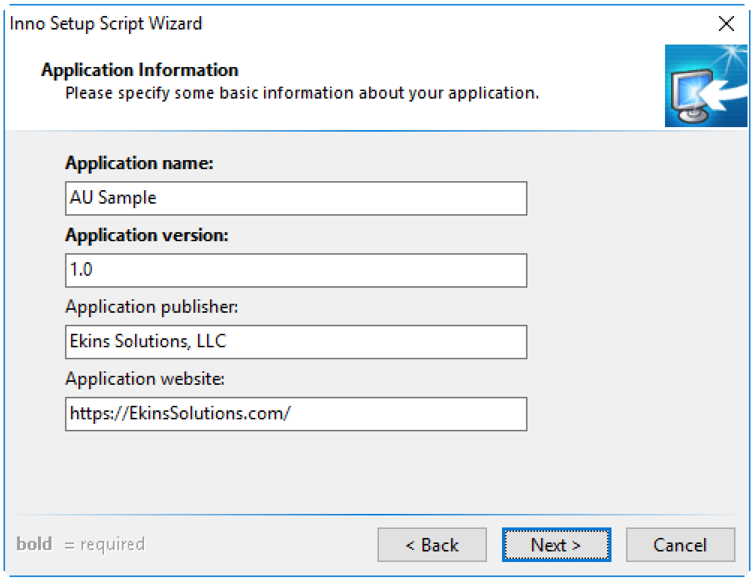
This wizard will guide you through the process of creating a new Inno Setup script file. The results will be used to generate a new script file which can be compiled directly or saved on disk for later use.

Not all features of Inno Setup are covered by this wizard. See the documentation for details on creating Inno Setup script files.

Click Next to continue, or Cancel to exit this wizard.

□ Create a new empty script file

4. Enter values for the **Application Information** page, as shown below. The publisher and website are optional and can be blank.



Inno Setup Script Wizard

Application Information

Please specify some basic information about your application.

Application name: AU Sample

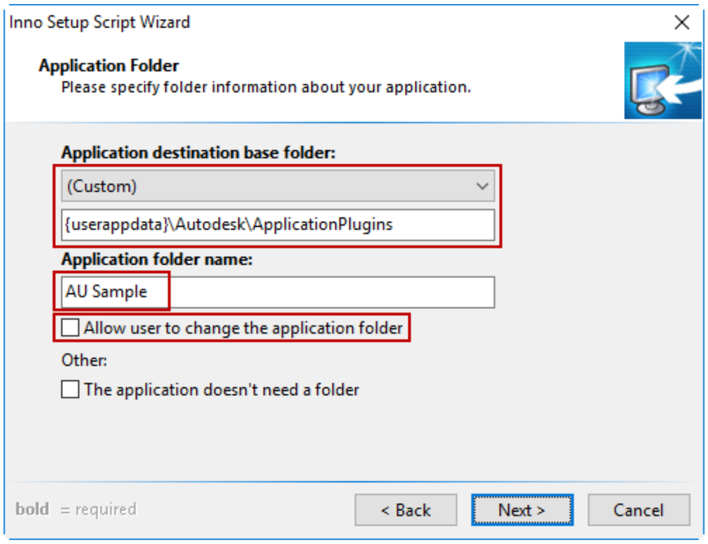
Application version: 1.0

Application publisher: Ekins Solutions, LLC

Application website: https://EkinsSolutions.com/

bold = required

5. Edit the **Application Folder** page as shown below, I changed the base folder to "(Custom)" and entered "{userappdata}\Autodesk\ApplicationPlugins" as the destination path. That's where the files will be installed. I specified "AU Sample" as the name of the folder that will be created and the files installed to. This is typically the name of your add-in. Finally, I unchecked the "Allow user to change the application folder" so the user can't change the install location.



Inno Setup Script Wizard

Application Folder
Please specify folder information about your application.

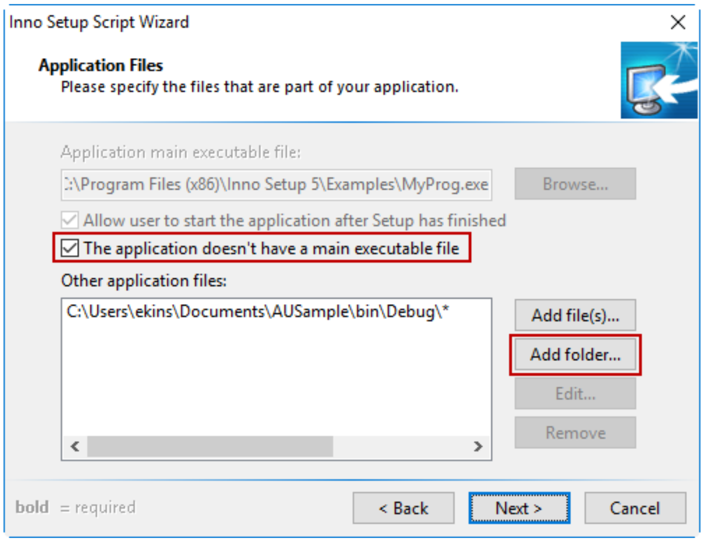
Application destination base folder:
(Custom)
{userappdata}\Autodesk\ApplicationPlugins

Application folder name:
AU Sample

□ Allow user to change the application folder

Other:
□ The application doesn't need a folder

6. For the **Application Files** page, check the box for "The application doesn't have a main executable file" and then click the "Add folder..." button, browse to your projects folder and choose the bin\Debug folder. Answer "Yes" to the question about including subfolders.



Inno Setup Script Wizard

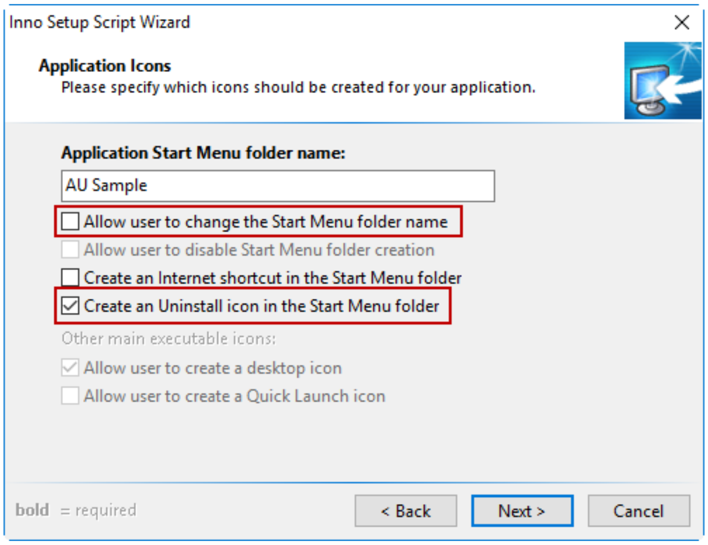
Application Files
Please specify the files that are part of your application.

Application main executable file:
:\Program Files (x86)\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\ekins\Documents\AUSample\bin\Debug\*

7. For the **Application Icons** page, uncheck “Allow user to change the Start Menu folder name and check “Create an Uninstall icon in the Start Menu folder”.



Inno Setup Script Wizard

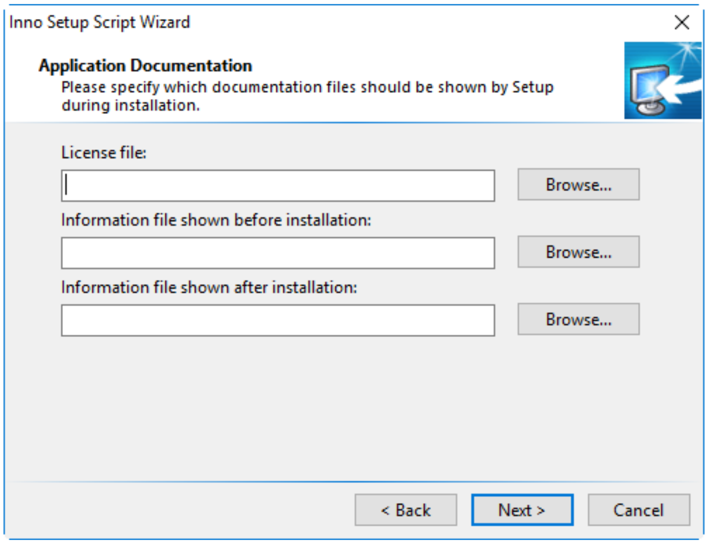
Application Icons
Please specify which icons should be created for your application.

Application Start Menu folder name:
AU Sample

☐ Allow user to change the Start Menu folder name
☐ Allow user to disable Start Menu folder creation
☐ Create an Internet shortcut in the Start Menu folder
☑ Create an Uninstall icon in the Start Menu folder

Other main executable icons:
☑ Allow user to create a desktop icon
☐ Allow user to create a Quick Launch icon

8. I didn’t change anything on the **Application Documentation** page, but you can specify license and information files if you want.



Inno Setup Script Wizard

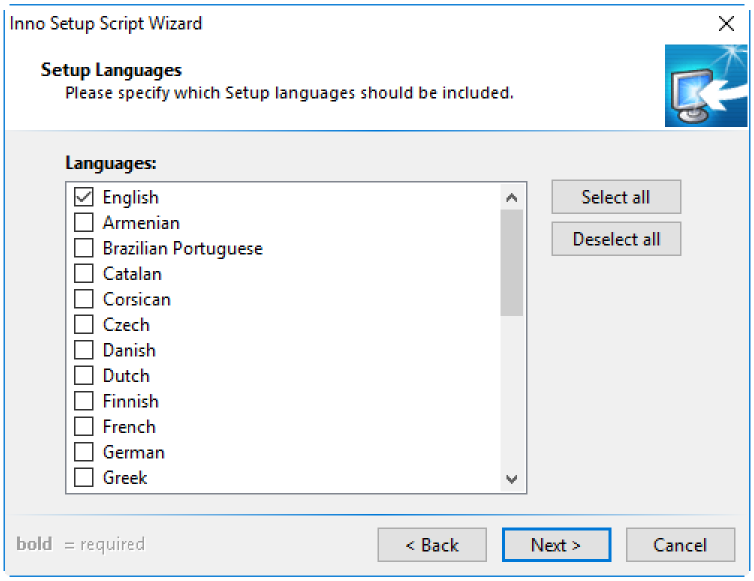
Application Documentation
Please specify which documentation files should be shown by Setup during installation.

License file:

Information file shown before installation:

Information file shown after installation:

9. I took the default “English” on the Setup Languages page. This defines the language of the installer, not your add-in.



Inno Setup Script Wizard

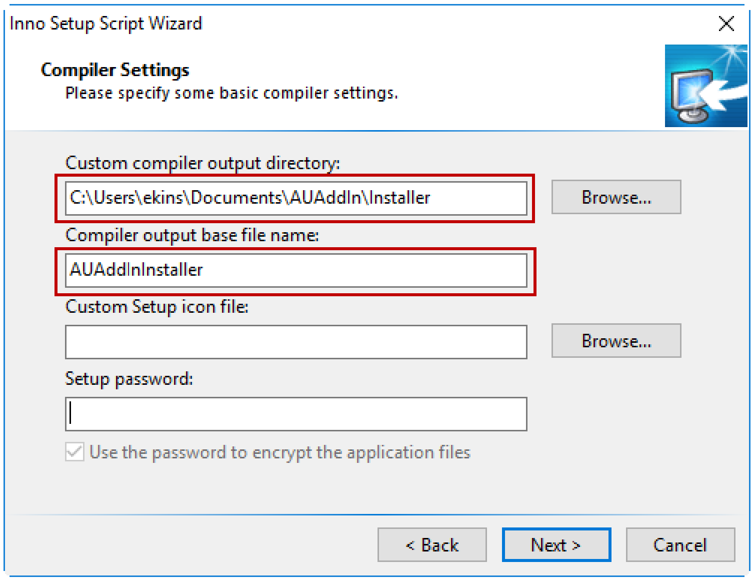
Setup Languages

Please specify which Setup languages should be included.

Languages:

☑ English
□ Armenian
□ Brazilian Portuguese
□ Catalan
□ Corsican
□ Czech
□ Danish
□ Dutch
□ Finnish
□ French
□ German
□ Greek

10. On the **Compiler Settings** page, specify the directory where the installer will be created and the name of the setup file. I created a subdirectory called “Installer” within my Visual Studio project folder. That way I can keep all the code related to the add-in together.



Inno Setup Script Wizard

Compiler Settings

Please specify some basic compiler settings.

Custom compiler output directory: C:\Users\ekins\Documents\AUAddIn\Installer

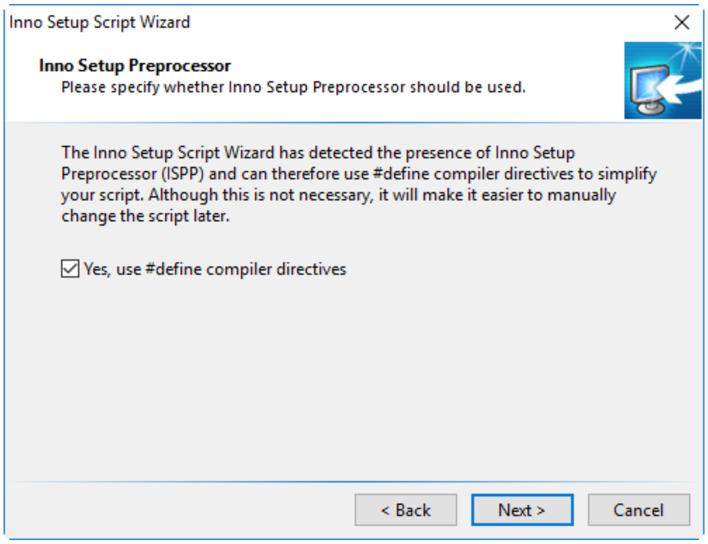
Compiler output base file name: AUAddInInstaller

Custom Setup icon file:

Setup password:

☑ Use the password to encrypt the application files

11. Take the default for the **Inno Setup Preprocessor** page.



Inno Setup Script Wizard

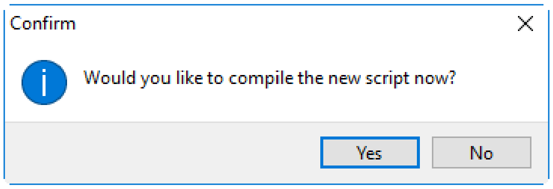
Inno Setup Preprocessor

Please specify whether Inno Setup Preprocessor should be used.

The Inno Setup Script Wizard has detected the presence of Inno Setup Preprocessor (ISPP) and can therefore use #define compiler directives to simplify your script. Although this is not necessary, it will make it easier to manually change the script later.

☑ Yes, use #define compiler directives

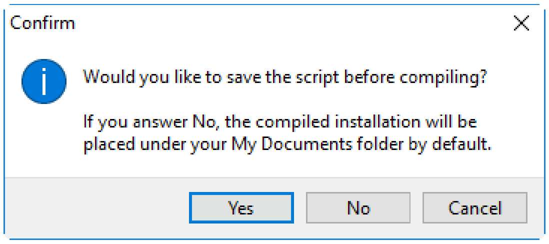
12. Click **Finish** which will finish the wizard and take you back to the Inno Script Studio interface and it will ask if you want to compile the script. Click “Yes” to create your installer.



Confirm

Would you like to compile the new script now?

This will result in the dialog below. Click “Yes” and choose the same folder that you specified in step 10. I enter the name of my add-in as the name of the .iss file that will be saved. The .iss file is the source for your installer project.

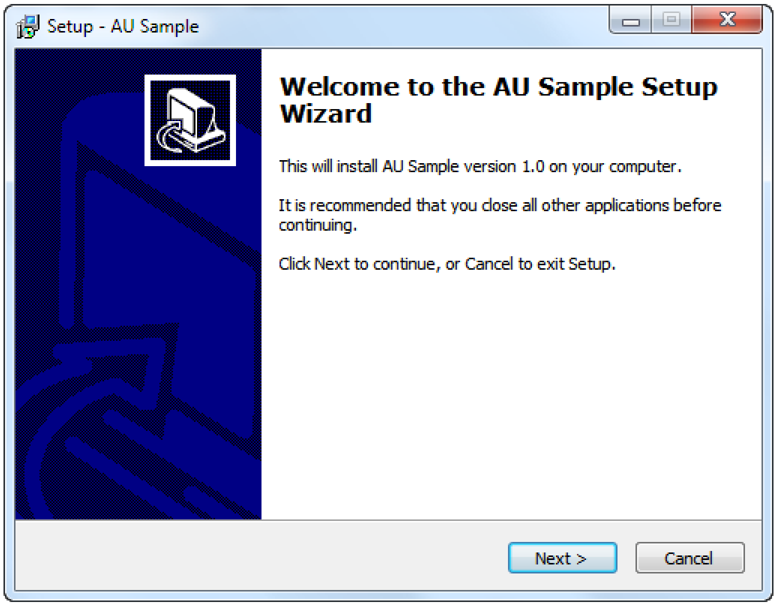


Confirm

Would you like to save the script before compiling?

If you answer No, the compiled installation will be placed under your My Documents folder by default.

13. Anyone can now install your add-in on their computer by running the installer you just created. In this example it is AUAddInInstaller.exe. They'll get the familiar installer sequence and can uninstall it using the standard Windows uninstall from the Control Panel.



Welcome to the AU Sample Setup Wizard

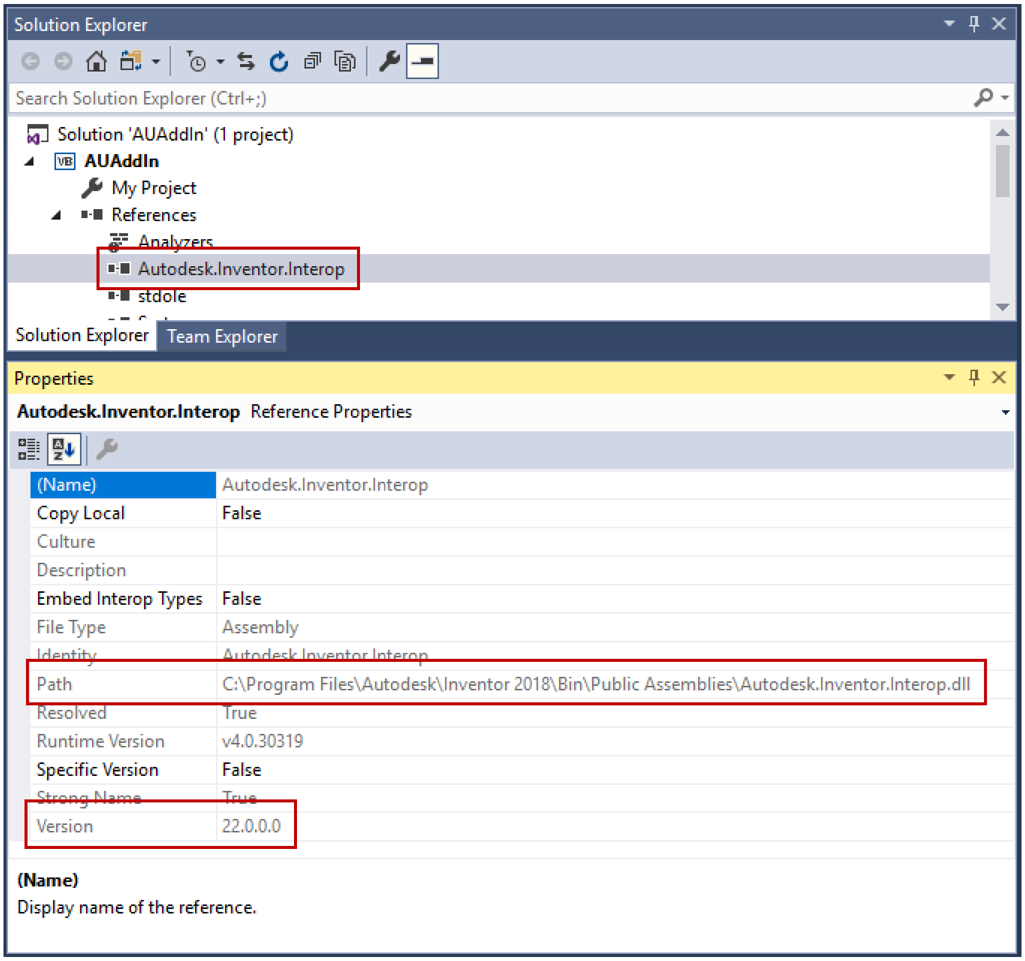
This will install AU Sample version 1.0 on your computer.

It is recommended that you close all other applications before continuing.

Click Next to continue, or Cancel to exit Setup.

Updating Your Add-In to a New Version of Inventor

An EXE or an add-in is loosely coupled to Inventor. The only tie to Inventor is that your project is referencing a file (.NET Interop) that describes the Inventor API. That's how Visual Studio can provide code completion. The Nifty add-in template creates a project that is referencing the interop for Inventor 2018. You can see this for your project in the Solution Explorer, as shown below. When I select "Autodesk.Inventor.Interop" from the list of references, information about the reference is displayed in the Properties window. In the path I can see "Inventor 2018" and the version is 22.0.0, which also corresponds with Inventor 2018.

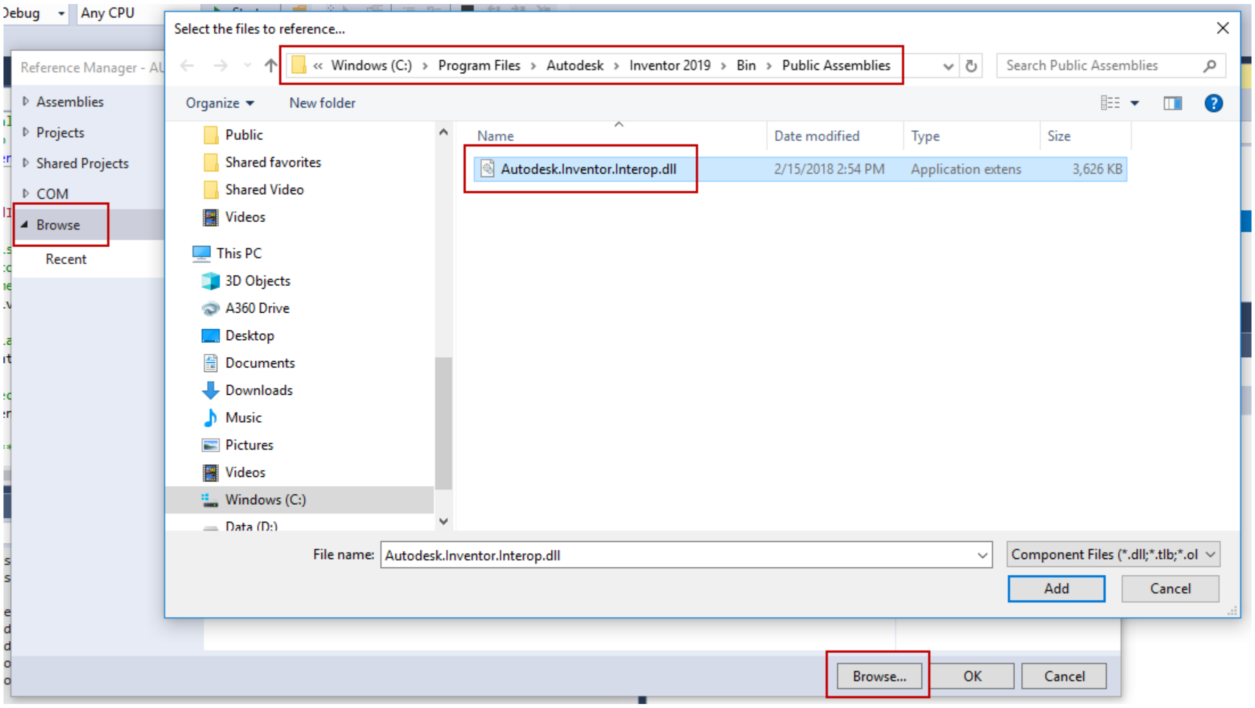


Autodesk.Inventor.Interop Reference Properties

(Name)	Autodesk.Inventor.Interop
Copy Local	False
Culture	
Description	
Embed Interop Types	False
File Type	Assembly
Identity	Autodesk.Inventor.Interop
Path	C:\Program Files\Autodesk\Inventor 2018\Bin\Public Assemblies\Autodesk.Inventor.Interop.dll
Resolved	True
Runtime Version	v4.0.30319
Specific Version	False
Strong Name	True
Version	22.0.0.0

(Name)
Display name of the reference.

What this means is that the add-in knows about the API that is delivered as part of Inventor 2018. If I use the add-in with Inventor 2019 it will still work. The interops for older versions of Inventor are delivered with Inventor to provide backward compatibility. The limitation will be if you write a new add-in and want to use new API functionality that's been added in Inventor 2019. Visual Studio won't know about the new functionality because the project is referencing the interop for 2018. To fix this, delete the current reference to Autodesk.Inventor.Interop by deleting the node in the Solution Explorer and then add a new reference by right-clicking on the References node and choosing "Add Reference…". There are different ways to add references to different kinds of libraries. For the Inventor interops, choose “Browse” from the panels on the left and then click the “Browse…” button at the bottom of the dialog. Next, find the Autodesk.Inventor.Interop.DLL file for the version of Inventor you want to use. For Inventor 2019, it’s in the “C:\Program Files\Autodesk\Inventor 2019\Bin\Public Assemblies” folder. You can modify the references of any existing project using this process.



Select the files to reference...

« Windows (C:) › Program Files › Autodesk › Inventor 2019 › Bin › Public Assemblies

Name	Date modified	Type	Size
Autodesk.Inventor.Interop.dll	2/15/2018 2:54 PM	Application extens	3,626 KB

File name: Autodesk.Inventor.Interop.dll

Component Files (*.dll;*.tlb;*.ol

Just remember that it isn’t necessary in most cases and is only needed when you need to use new API functionality. In fact, it’s usually better to use an older interop so your program will be compatible with more versions of Inventor. For example, if I reference the interop for Inventor 2017 into my add-in, my add-in will work with Inventor 2017 and all later versions. It won’t work with versions before Inventor 2017. If I write it for Inventor 2019, it won’t work with earlier versions.