Official Guide

Add-in Express™ .net

Getting Started

Add-in Express™ 2010 for Microsoft® Office and .NET

Revised on 26-Jul-10

Table of Contents

Introduction

Add-in Express is a development tool designed to simplify and speed up the development of Office COM Add-ins, Run-Time Data servers (RTD servers), Smart Tags, Excel Automation Add-ins and Excel XLL add-ins in Visual Studio 2005-2010 through the consistent use of the RAD paradigm. It provides a number of specialized components allowing the developer to skip the interface-programming phase and get to functional programming in no time.

Why Add-in Express?

Add-in Express and Office Extensions

Microsoft introduced the term Office Extensions. This term covers all the customization technologies provided for Office applications. The technologies are:

- COM Add-ins
- Smart Tags
- Excel RTD Servers
- Excel Automation Add-ins
- Excel XLL Add-ins

Add-in Express allows you to overcome the basic problem when customizing Office applications in .NET – building your solutions into the Office application. Based on the True RAD paradigm, Add-in Express saves the time that you would have to spend on research, prototyping, and debugging numerous issues of any of the above-said technologies in all versions and updates of all Office applications. The issues include safe loading / unloading, host application startup / shutdown, as well as user-interaction and deployment issues.

Add-in Express provides you with simple tools for creating version-neutral, secure, insolated, managed, deployable, and updatable Office extensions.

- Managed Office Extensions

You develop them in every programming language available for Visual Studio .NET (see System Requirements).

- Isolated Office Extensions

Add-in Express allows loading Office extensions into separate application domains. Therefore, the extensions do not have a chance to break down other add-ins and the host application itself. See How Your Office Extension Loads Into an Office Application.

- Version-neutral Office Extensions

The Add-in Express programming model and its core are version-neutral. That is, you can develop one Office extension for all available Office versions, from 2000 to the newest 2010. See Choosing Interop Assemblies.

- Deployable and updatable Office Extensions

Add-in Express automatically supplies you with a setup project making your solution ready-to-deploy. The start-up and deployment model used by Add-in Express allows updating your solutions at run-time See also Deploying Add-in Express Projects

Add-in Express Products

Add-in Express provides a number of products for developers on its web site.

- Add-in Express 2010 for Microsoft Office and CodeGear VCL

It allows creating fast version-neutral native-code COM add-ins, smart tags, Excel automation add-ins, and RTD servers in Delphi. See http://www.add-in-express.com/add-in-delphi/.

- Add-in Express 2010 for Internet Explorer and .NET

It allows developing add-ons for IE 6, 7 and 8 in .NET. Custom toolbars, sidebars and BHOs are already on board. See http://www.add-in-express.com/programming-internet-explorer/.

- Security Manager 2010 for Microsoft Outlook

This is a product designed for Outlook solution developers. It allows controlling the Outlook e-mail security guard by turning it off and on in order to suppress unwanted Outlook security warnings. See http://www.add-in-express.com/outlook-security/.

System Requirements

Add-in Express supports developing Office extensions in VB.NET, C# and C++.NET on all editions of VS 2005, 2008 and 2010.

C++ .NET isn't supported in Express editions of Visual Studio 2005-2010.

Host Applications

COM Add-ins

- Microsoft Excel 2000 and higher
- Microsoft Outlook 2000 and higher
- Microsoft Word 2000 and higher
- Microsoft FrontPage 2000 and higher
- Microsoft PowerPoint 2000 and higher
- Microsoft Access 2000 and higher
- Microsoft Project 2000 and higher
- Microsoft MapPoint 2002 and higher
- Microsoft Visio 2002 and higher
- Microsoft Publisher 2003 and higher
- Microsoft InfoPath 2007 and higher

Real-Time Data Servers

- Microsoft Excel 2002 and higher

Smart Tags

- Microsoft Excel 2002 and higher
- Microsoft Word 2002 and higher
- Microsoft PowerPoint 2003 and higher

Smart tags are deprecated in Excel 2010 and Word 2010. Though, you can still use the related APIs in projects for Excel 2010 and Word 2010, see Changes in Word 2010 and Changes in Excel 2010.

Excel Automation Add-ins

- Microsoft Excel 2002 and higher

Excel XLL Add-ins

- Microsoft Excel 2000 and higher

Technical Support

Add-in Express is developed and supported by the Add-in Express Team, a branch of Add-in Express Ltd. The Add-in Express web site at www.add-in-express.com provides a wealth of information and software downloads for Add-in Express developers, including:

- Our technical blog provides the most recent information as well as How To and Video How To samples.
- The HOWTOs section contains sample projects answering most common "how to" questions.
- Add-in Express Toys contains "open sourced" add-ins for popular Office applications.
- Built-in Controls Scanner utility: find IDs of built-in CommandBar controls. It is free.
- MAPI Store Accessor – this is a .NET wrapper over Extended MAPI. It is free, too.

For technical support through the Internet use our forums or e-mail us at support@add-in-express.com. We are actively participating in these forums.

If you are a subscriber of our Premium Support Service and need help immediately, you can request technical support via an instant messenger, e.g. Windows/MSN Messenger or Skype.

Installing and Activating

There are two main points in the Add-in Express installation. First off, you have to specify the development environments in which you are going to use Add-in Express (see System Requirements). Second, you need to activate the product.

Activation Basics

During the registration process, the registration wizard prompts you to enter your license key. The key is a 30 character alphanumeric code shown in six groups of five characters each (for example, AXN4M-GBFTK-3UN78-MKF8G-T8GTY-NQS8R). Keep the license key in a safe location and do not share it with others. This product key forms the basis for your ability to use the software.

For purposes of product activation only, a non-unique hardware identifier is created from general information that is included in the system components. At no time are files on the hard drive scanned, nor is personally identifiable information of any kind used to create the hardware identifier. Product activation is completely anonymous. To ensure your privacy, the hardware identifier is created by what is known as a "one-way hash". To produce a one-way hash, information is processed through an algorithm to create a new alphanumeric string. It is impossible to calculate the original information from the resulting string.

Your product key and a hardware identifier are the only pieces of information required to activate the product. No other information is collected on your PC or sent to the activation server.

If you choose the *Automatic Activation Process* option of the activation wizard, the wizard attempts to establish an online connection to the activation server, www.activatenow.com. If the connection is established, the wizard sends both the license key and the hardware identifier over the Internet. The activation service generates an activation key using this information and sends it back to the activation wizard. The wizard saves the activation key to the registry.

If an online connection cannot be established (or you choose the *Manual Activation Process* option), you can activate the software using your web-browser. In this case, you will be prompted to enter the product key and a hardware identifier on a web page, and will get an activation key in return. This process finishes with saving the activation key to the registry.

Activation is completely anonymous; no personally identifiable information is required. The activation key can be used to activate the product on that computer an unlimited number of times. However, if you need to install the product on several computers, you will need to perform the activation process again on every PC. Please refer to your end-user license agreement for information about the number of computers you can install the software on.

Setup Package Contents

The Add-in Express 2010 for .NET setup program installs the following folders on your PC:

- Bin – Add-in Express binary files
- Docs – Add-in Express documentation including class reference
- Images – Add-in Express icons
- Redistributables – Add-in Express redistributable files including interop assemblies, see Redistributables
- Sources – Add-in Express source code (see the note below)

Please note that the source code of Add-in Express is or is not delivered depending on the product package you purchased. See the Feature matrix and prices page on our web site for details.

Add-in Express setup program installs the following text files on your PC:

- licence.txt – EULA
- readme.txt – short description of the product, support addresses and such
- whatsnew.txt – this file describes the latest information on the product features added and bugs fixed.

Solving Installation Problems

Make sure you are an administrator on the PC. On Vista, Windows 7 and Windows 2008 Server, set UAC to its default level. In Control Panel | System | Advanced | Performance | Settings | Data Execution Prevention, set the "... for essential Windows programs and services only" flag. Remove the following registry key, if it exists:

```
HKEY_CURRENT_USER\Software\Add-in Express\{product identifier} {version} {package}
```

Run setup.exe, not .MSI. Finally, use the Automatic activation option in the installer windows.

Redistributables

See {Add-in Express}\Redistributables. You will find a readme.txt in that folder.

Several redistributable files are located in {Add-in Express}\Bin. Here are their descriptions:

File name	Description
AddinExpress.MSO.2005.dll	Office + XLL add-ins + Excel Automation add-ins
AddinExpress.RTD.2005.dll	RTD servers
AddinExpress.SmartTag.2005.dll	Smart tags
AddinExpress.OL.2005.dll	Advanced Outlook form regions
AddinExpress.PP.2005.dll	Advanced Office task panes in PowerPoint
AddinExpress.WD.2005.dll	Advanced Office task panes in Word
AddinExpress.XL.2005.dll	Advanced Office task panes in Excel
AddinExpress.ToolbarControls.2005.dll	.NET controls on Office command bars

Getting Started

In this chapter, we guide you through the following steps of developing Add-in Express projects:

- Creating an Add-in Express project
- Adding an Add-in Express designer to the project
- Adding Add-in Express components to the designer
- Adding some business logics
- Building, registering, and debugging the Add-in Express project
- Tuning up the Add-in Express loader based setup project
- Deploying your project to a target PC

What's New in Add-in Express 2010

Support for Office 2010 32-bit and 64-bit including new features of Ribbon in Office 2010

Support for Visual Studio 2010

VS 2003 isn't supported any longer

New project wizard

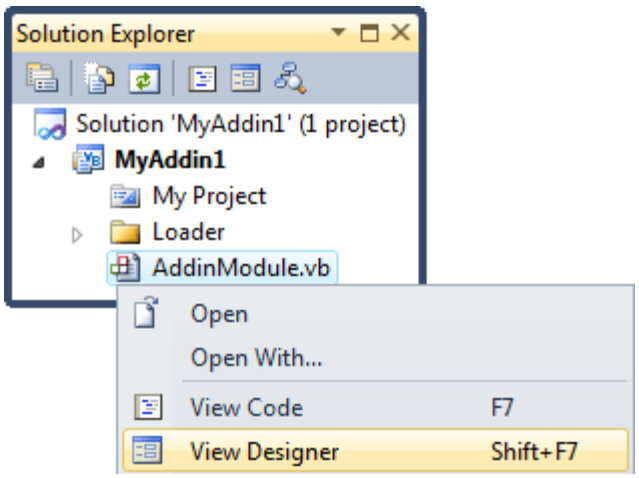
New setup project wizard (see also Modifying an existing setup project to support Office 2010, 32-bit and 64-bit, in Add-in Express 2010)

New deployment technology (see Web-based MSI deployment)

Add-in Express Basics

Modules

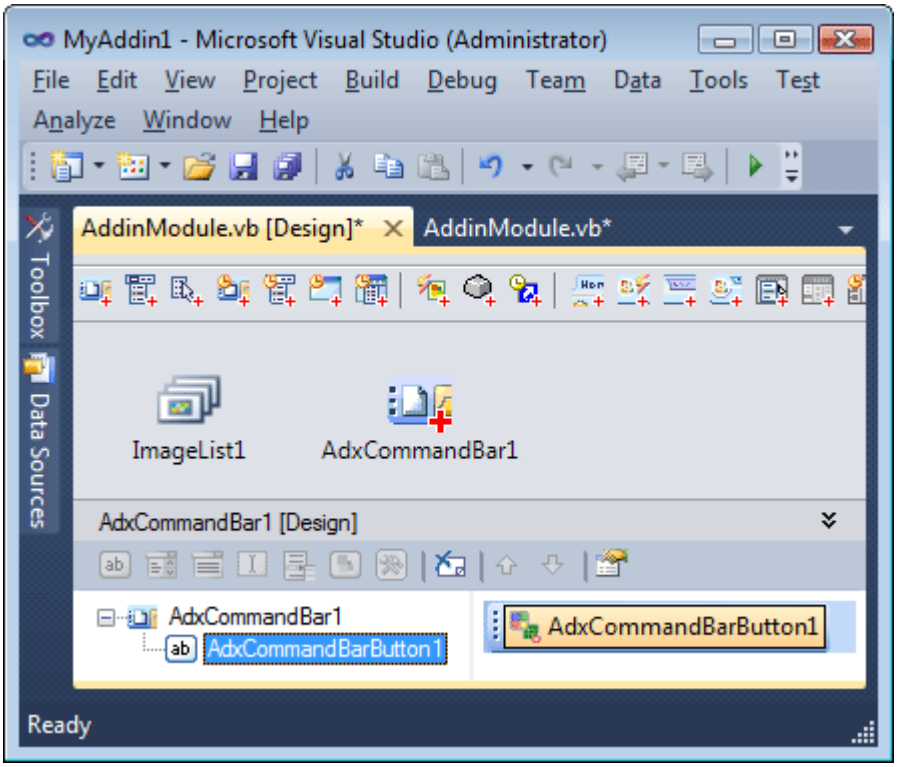
An Add-in Express based project (see New Project dialog) includes a file called *module*. The module is the core component of the project; it represents a COM add-in, Excel UDF, etc. Each module provides a designer accessible in a number of ways. See one of them in the screenshot.



The designer surface of the module incorporates:

- **Commands** toolbar - it allows adding Add-in Express components available for the module.
- **Components** area - it is what a usual designer provides
- **In-place designer** area - if the visual designer for the currently selected Add-in Express component is available, it is shown in this area

The screenshot below shows the add-in module designer with the in-place designer for the ADXCommandBar component. See also Add-in Express Components and Sample Projects.



Host Application UI

Add-in Express provides a number of components that allow customizing both the command bar UI and Ribbon UI of Office applications. Creating advanced task panes in Outlook, Word, Excel and PowerPoint version 2000-2010 is also supported. See Add-in Express Components.

Host Application Events

Add-in Express provides a number of application-specific objects that allow specifying event handlers for application-level events of all Office applications. To handle such events, you need to add an Events object such as Outlook Events or Excel Events to the add-in module and specify event handlers for required events. See Events.

In addition, Add-in Express supplies events classes providing methods in which you write your code to handle this or that event declared in a host application object other than the Application object of the host application. See Events.

Supporting Several Office Versions in the Same Project

There are two aspects of this theme:

- Supporting the CommandBar and Ribbon UI in one project

You can add both Command Bar UI and Ribbon UI components onto the add-in module. When your add-in is loaded in a particular version of the host application, either command bar or ribbon controls will show up. Find additional information in Command Bars in the Ribbon UI.

- Accessing version-specific features of an Office application

Please see Choosing Interop Assemblies.

Developing Multiple Office Extensions in the Same Project

Add-in Express supports adding several modules (see Add-in Express Basics) in a project. That means you can create an assembly containing a combination of several Office extensions. Having several modules in an assembly is a common approach to developing Excel extensions; say you can implement a COM add-in providing some settings for your Excel UDF.

What is essential is that all Office extensions will be loaded into the same AppDomain. The only exception is Excel Automation add-ins – they are loaded into the Default AppDomain (but see What Excel UDF Type to Choose?).

If several Office extensions are gathered in one assembly, Office loads the assembly once but initializes the extensions in the assembly one at a time. That is, if you have two COM add-ins in the same assembly, one of them may be still not initialized when the first one is ready to work. See also HowTo: Create a COM add-in, XLL UDF and RTD server in one assembly.

See also Deploying Office Extensions and Accessing Public Members of Your COM Add-in from Another Add-in or Application

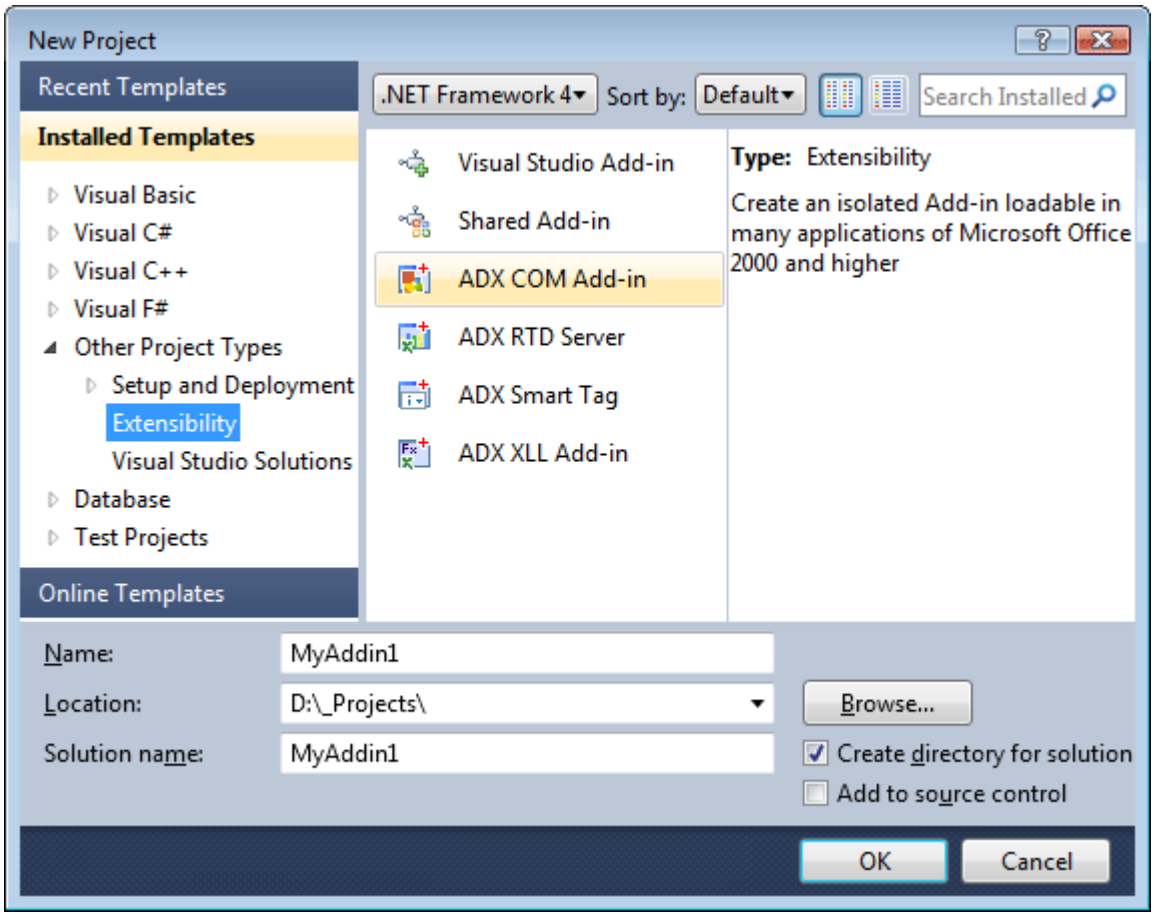
Creating Add-in Express Projects

Add-in Express installs a number of items to the New Project dialog as well as to the Add New Item dialog in order to allow creating the following customization types:

- COM Add-ins
- Excel RTD Servers
- Smart Tags
- Excel UDFs (including Excel Automation Add-ins and Excel XLL Add-ins)
- Excel Workbooks
- Word Documents

New Project dialog

Add-in Express adds several project templates to the Extensibility folder of the *New Project* dialog in Visual Studio. To see the dialog, choose *File* | *New* | *Project…* in the main menu.



Whichever Add-in Express project template you choose, it starts a project wizard that allows selecting a programming language for your project, interop assemblies to use as well as other options. The project wizard creates a new solution containing an appropriate Add-in Express project.

Each Add-in Express project contains an appropriate designer class also called "module": add-in module, XLL module, RTD server module, etc. Project-specific modules are the core components of Add-in Express. You can add any components onto the modules. Add-in Express provides a number of components that simplify and speed up the development of Office extensions, see Add-in Express Components.

Choosing Interop Assemblies

An Office interop assembly provides the compiler with early-binding information on COM interfaces contained in a given Office application (COM library) of a given version. That's why there are interops for Office 2003, 2007, etc.

Because Office applications are almost 100% backward compatible, you can still use any interop version to access any version of the host application. There are two things worth mentioning:

When using an interop for an arbitrary Office version, you are required to check the version of the Office application that loads your add-in before accessing two kinds of things: a) that introduced in a newer Office version and b) that missing in an older Office version.

For instance, consider developing an Outlook add-in using the Outlook 2003 interop; the add-in must support Outlook 2000 - 2010. Let's examine accessing two properties of the MailItem class: a) MailItem.Sender introduced in Outlook 2010 and b) MailItem.BodyFormat introduced in Outlook 2002.

Since that add-in uses the Outlook 2003 interop, you cannot just write sender = theMailItem.Sender in your code: doing this will cause a compile-time error. To bypass this, you must write a code that checks if the add-in is loaded in Outlook 2010 and use late binding to access that property. "Late binding" means that you use Type.InvokeMember(); look at this article on MSDN or search for samples on our .NET forum.

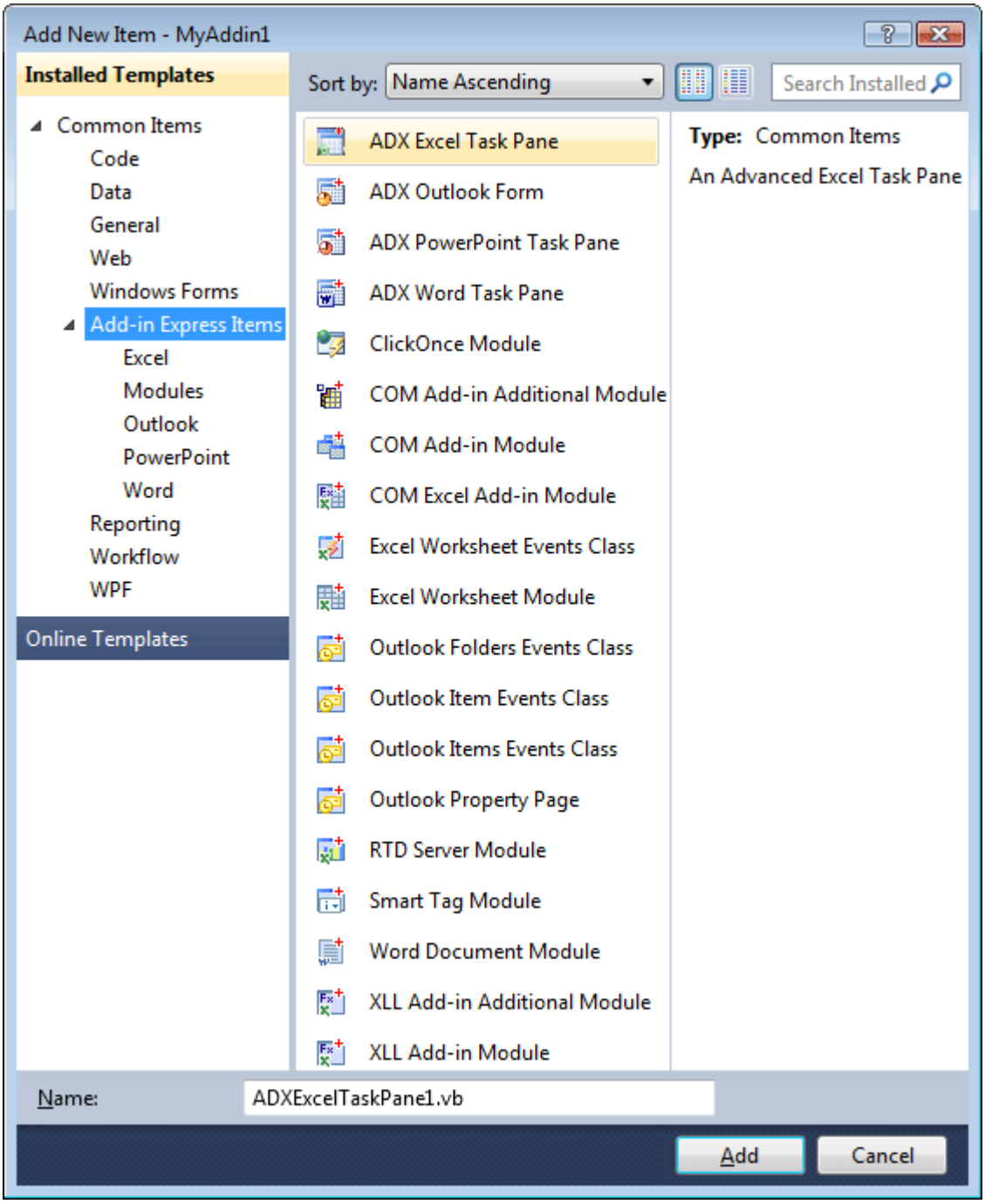
Since MailItem.BodyFormat is missing in Outlook 2000, you cannot just write bodyFormat = theMailItem.BodyFormat: doing this will fire a run-time exception when your add-in is loaded in Outlook 2000. To bypass this, you must write a code that checks if the add-in is loaded in Outlook 2000 and avoid accessing that property in this case.

The following questions are discussed in this article on our blog:

- What interop assembly to choose for your add-in project?
- How does an interop assembly version influence the development time?
- How to support a given Office version correctly?

Add New Item dialog

Add-in Express installs the following items to the *Add New Item* dialog (right-click your project item in *Solution Explorer* and choose *Add* | *New Item* in the context menu).



- *Add-in Express Excel Task Pane* – a form designed for being embedded into Excel windows. See Advanced Custom Task Panes, Excel Task Panes and Your First Microsoft Office COM Add-in.

- *Add-in Express Outlook Form* – a form designed for being embedded into Outlook Explorer and Inspector windows. See Advanced Custom Task Panes, Advanced Outlook Regions and Your First Microsoft Outlook COM Add-in.
- *Add-in Express PowerPoint Task Pane* – a form designed for being embedded into PowerPoint. See Advanced Custom Task Panes and Your First Microsoft Office COM Add-in.
- *Add-in Express Word Task Pane* – a form designed for being embedded into Word documents. See Advanced Custom Task Panes and Your First Microsoft Office COM Add-in.
- *ClickOnce Module* – allows accessing ClickOnce-related features in ClickOnce Deployment.
- *COM Add-in Additional Module* – it is an additional add-in module. See How to Develop the Modular Architecture of your COM and XLL Add-in?
- *COM Add-in Module* – the core of any Add-in Express based COM add-in. See COM Add-ins, Your First Microsoft Office COM Add-in and Your First Microsoft Office COM Add-in.
- *COM Excel Add-in Module* – this module allows implementing user-defined functions in Excel. See Excel UDFs and Your First Excel Automation Add-in.
- *Excel Worksheet Events Class* – provides easy access to the events of the Worksheet class. See Your First Microsoft Office COM Add-in.
- *Excel Worksheet Module* – allows handling events of any MS Forms controls placed on a specified Excel worksheet. See Excel Workbooks.
- *Outlook Folders Events Class* – provides easy access to the events of the Folders class of Outlook. See Events Classes.
- *Outlook Item Events Class* – provides easy access to the events of the MailItem, TaskItem, ContactItem, etc classes of Outlook. See Events Classes.
- *Outlook Items Events Class* – provides easy access to the events of the Items class of Outlook. See Events Classes and Your First Microsoft Outlook COM Add-in.
- *Outlook Property Page* – the form designed for extending *Outlook Options* and *Folder Properties* dialogs with custom pages. See Outlook Property Page and Your First Microsoft Outlook COM Add-in.
- *Smart Tag Module* – the core of an Add-in Express based smart tag. See Smart Tags and Your First Smart Tag.
- *RTD Server Module* – the core of an Add-in Express based RTD server. See Excel RTD Servers and Your First Excel RTD Server.
- *Word Document Module* – allows handling events of any MS Forms controls placed on a specified Word document. See Word Documents.
- *XLL Add-in Additional Module* – it is an additional XLL add-in module. See How to Develop the Modular Architecture of your COM and XLL Add-in?
- *XLL Add-in Module* – allows developing user-defined functions in Excel. See Excel UDFs and Your First XLL add-in.

COM Add-ins

COM add-ins have been around since Office 2000 when Microsoft allowed Office applications to extend their features with COM DLLs supporting the IDTExtensibility2 interface (it is a COM interface, of course). Since then thousands of developers have racked their brains over this interface and the Office object model that provided COM objects representing command bars, command bar controls, etc. These were the sources of Add-in Express.

Why COM add-ins?

COM add-ins is the only way to provide new or re-use built-in UI elements such as command bar controls and Ribbon controls. Say, a COM add-in can show a command bar or Ribbon button to process selected Outlook e-mails, Excel cells, or paragraphs in a Word document and perform some actions on the selected objects. A COM add-in supporting Outlook, Excel, Word or PowerPoint can show custom task panes in Office 2007 and higher and Add-in Express panes in Office 2000-2010, see Task Panes. In a COM add-in targeting Outlook, you can add custom option pages to the *Tools* | *Options* and *Folder Properties* dialogs (see Step #14 – Adding Property Pages to the Folder Properties Dialog). A COM add-in also handles events and calls properties and methods provided by the object model of the host application. For instance, a COM add-in can modify an e-mail when it is being sent; it can cancel saving an Excel workbook; or, it can check if a Word document meets some conditions.

Per-user and per-machine COM add-ins

A COM add-in can be registered either for the current user (the user running the installer) or for all users on the machine. That's why the corresponding module type, ADXAddinModule, provides the RegisterForAllUsers property. Registering for all users means writing to HKLM and that means the user registering a per-machine add-in must have administrative permissions. Accordingly, RegisterForAllUsers = Flase means writing to HKCU (=for the current user).

See also Registry Keys.

An add-in deployed via ClickOnce can be registered with HKCU only. See also ClickOnce Deployment.

A standard user may turn a per-user add-in off and on in the COM Add-ins Dialog. You use that dialog as well as the *{host application}* | *Options* | *Add-ins* dialog in Office 2007-2010 to find if your add-in is active.

Creating a COM Add-in project

To create a COM add-in, choose the *Add-in Express COM Add-in* project template in the New Project dialog. The core of the project is the add-in module, of the ADXAddinModule type. The add-in module represents a COM add-in in any Office application (see Host Applications). To add another add-in to your assembly, add another add-in module to your project (see Add New Item dialog and Architecture). For the add-in, you specify its name, host application(s) and load behavior. The typical value for the LoadBehavior

property is *Connected & LoadAtStartup*. That value is written to the registry (see Registry Keys) when you register the add-in.

For Outlook add-ins, you also specify pages for the *Tools* | *Options* and *Folder Properties* dialogs (see Outlook Property Page).

See the following chapters for the Add-in Express components you add onto add-in modules: Ribbon UI, Command Bar UI, Connecting to Existing CommandBar Controls, Intercepting Keyboard Shortcuts, Advanced Custom Task Panes, Outlook Bar Shortcut Manager, Application-level Events.

Pay attention to the AddinExpress.MSO.ADXAddinModule.CurrentInstance method (it's static in C#, Shared in VB.NET); it allows accessing public properties and method outside of the module.

Use the AddinStartupComplete and AddinBeginShutdown events to handle add-in startup and shutdown.

This guide describes two sample add-in projects: see Your First Microsoft Office COM Add-in and Your First Microsoft Outlook COM Add-in.

What's next?

You need to study the following areas before implementing the business logic of your add-in:

- Creating the UI of your add-in and handling events of the host application, see Add-in Express Components and Sample Projects.
- Deploying and updating your add-in, see Deploying Add-in Express Projects

Also, find useful information in Add-in Express Tips and Notes.

Excel RTD Servers

RTD Server is a technology introduced in Excel XP.

Why RTD server?

An RTD server is used to provide the end user with a flow of changing data such as stock quotes, currency exchange rates etc. If an RTD server is mentioned in a formula (placed on an Excel worksheet), Excel loads the RTD server and waits for new data from it. When data arrive, Excel seeks for a proper moment and updates the formula with new data.

RTD Server terminology

- An RTD server is a Component Object Model (COM) Automation server that implements the IRtdServer COM interface. Excel uses the RTD server to communicate with a real-time data source on one or more topics.
- A real-time data source is any source of data that you can access programmatically.
- A topic is a string (or a set of strings) that uniquely identifies a data source or a piece of data that resides in a real-time data source. The RTD server passes the topic to the real-time data source and receives the value of the topic from the real-time data source; the RTD server then passes the value of the topic to Excel for displaying. For example, the RTD server passes the topic "New Topic" to the real-time data source, and the RTD server receives the topic's value of "72.12" from the real-time data source. The RTD server then passes the topic's value to Excel for display.

Per-user and per-machine RTD Servers

An RTD Server can be registered either for the current user (the user running the installer) or for all users on the machine. That's why the corresponding module type, ADXRTDServerModule, provides the RegisterForAllUsers property. Registering for all users means writing to HKLM and that means the user registering a per-machine RTD server must have administrative permissions. Accordingly, RegisterForAllUsers = Flase means writing to HKCU (=for the current user).

Creating an RTD server

To create an RTD server, choose *Add-in Express RTD Server* in the Add New Item dialog. The project designer type is ADXRtdServerModule (RTD server module). The only Add-in Express component allowed for this designer is RTD Topic. The module provides the Interval property that indicates the time interval between updates (in milliseconds).

You refer to an existing RTD Server using the RTD worksheet function in Excel:

```
=RTD(ProgID, Server, String1, String2, ... String28)
```

The ProgID parameter is a required string value representing the programmatic ID (or ProgID – see What is ProgID?) of the RTD server. See attributes of the RTDServerModule class for the ProgID of your RTD Server. The current version of Add-in Express requires the Server parameter to be an empty string. Use two quotation marks (""). The String1 through String28 parameters allow specifying topics of the RTD server. Only the String1 parameter is required; the String2 through String28 parameters are optional. The actual values for the String1 through String28 parameters depend on the requirements of the real-time data server.

What's next?

Please see RTD Topic. A sample project is described in Your First Excel RTD Server. Also, find useful information in How to Get Actual Parameters of the RTD function When Using an Asterisk in the String## Properties of a Topic? and Inserting the RTD Function in a User-Friendly Way.

Smart Tags

Office XP bestowed Smart Tags upon us in Word and Excel. Office 2003 added PowerPoint to the list of smart tag host applications. This technology provides Office users with more interactivity for the content of their Office documents. A smart tag is an element of text in an Office document having custom actions associated with it. Smart tags allow recognizing such text using either a dictionary-based or a custom-processing approach. An example of such text might be an e-mail address you type into a Word document or an Excel workbook. When smart tag recognizes the e-mail address, it allows the user to choose one of the actions associated with the text. For e-mail addresses, possible actions are to look up additional contact information or send a new e-mail message to that contact.

Smart tags are deprecated in Excel 2010 and Word 2010. Although you can still use the related APIs in projects for Excel 2010 and Word 2010, these applications do not automatically recognize terms, and recognized terms are no longer underlined. Users must trigger recognition and view custom actions associated with text by right-clicking the text and clicking the Additional Actions on the context menu. Please see Changes in Word 2010 and Changes in Excel 2010.

To create a smart tag, choose *Add-in Express Smart Tag* in the New Project dialog. ADXSmartTagModule, smart tag module, constitutes the base of Add-in Express smart tags. It represents a set or a library of smart tag recognizers in Excel, Word, and PowerPoint. The only Add-in Express component you add to the designer is Smart Tag.

See also Your First Smart Tag.

Excel UDFs

Excel UDFs are used to build custom functions in Excel for the end user to use them in formulas. This definition underlines the main restriction of an Excel UDF: it must return a result that can be used in a formula – not an object of any given type but a number, a string, or an error value (Booleans and dates are essentially numbers). When used in an array formula, the UDF must return a properly dimensioned array of values of the types above.

There are two Excel UDF types: Excel Automation Add-ins and Excel XLL Add-ins. They differ in several ways: see What Excel UDF Type to Choose?

What Excel UDF Type to Choose?

Automation add-ins are supported starting from Excel 2002; XLL add-ins work in Excel 2000 and higher.

Automation add-ins are suitable if your UDF deals a lot with the Excel object model; XLL add-ins are faster in financial and mathematical calculations. Note however that native code XLL add-ins work faster than managed UDFs.

Information below applies to the Add-in Express implementation of Excel Automation add-ins and XLL Add-ins.

When developing a combination of Excel extensions (see Developing Multiple Office Extensions in the Same Project), Add-in Express loads all of them into the same AppDomain. The only exception is the Excel Automation Add-in, which is loaded into the default AppDomain. You can bypass this by calling any public method of your Excel Automation add-in via ExcelApp.Evaluate(...) **before** Excel invokes the Automation add-in. ExcelApp.Evaluate(...) returns an error code if the Automation add-in isn't loaded; if it is the case, you need to call that method later, say in WorkbookActivate. We assume, however that this approach will not help in the general case. There's no such problem with XLL add-ins; they always load into the AppDomain shared by all Office extensions in your assembly.

An XLL add-in cannot have a description. The description of an Automation add-in is taken from the ProgId attribute applied to the Excel Add-in Module (of the ADXExcelAddinModule type). According to this page, ProgId is limited to 39 characters and can contain no punctuation other than a period.

You cannot hide a function in an Automation add-in. Moreover, in the *Insert Function* dialog, the user will see all public functions exposed by ADXExcelAddinModule, such as GetType and GetLifetimeService.In an XLL add-in, you hide a function by setting ADXExcelFunctionDescriptor.IsHidden=True, see Step #4 – Configuring UDFs.

Only functions (=methods returning a value) are acceptable in an Automation add-in. An XLL add-in may contain a procedure (=method, the return type of which is void); you can hide it in the UI (see above) and call it from say, a COM add-in, via ExcelApp.Evaluate(…).

XLL add-ins provide access to low-level Excel features through the ADXXLLModule.CallWorksheetFunction method; this method is a handy interface to functions exported by XLCALL32.DLL. No such feature is available for Automation add-ins.

In an Automation add-in, neither functions nor their arguments can have a description. For an XLL add-in, see Step #4 – Configuring UDFs. See also My XLL Add-in Doesn't Show Descriptions.

Excel Automation Add-ins

Excel 2002 brought in Automation Add-ins – a technology that allows writing user-defined functions for use in Excel formulas. Add-in Express reduces this task to just writing one or more user-defined functions. A

typical function accepts one or more Excel ranges and/or other parameters. Excel shows the resulting value of the function in the cell where the user calls the function.

To create an Excel Automation add-in, create a COM add-in project (see COM Add-ins) and choose *COM Excel Add-in Module in COM Add-in* in the Add New Item dialog. This adds an ADXExcelAddinModule (Excel add-in module) to the COM add-in project. The module represents an Excel Automation add-in. It does not provide any properties. See What's next?

Excel XLL Add-ins

An XLL is a DLL written in such a way that Excel can open it directly. Like Excel Automation add-ins, XLL add-ins are mostly used to create user-defined functions, however they work much faster. This technology was introduced in Excel 4.0; Wikipedia states that this happened in 1992! Since then, XLL interfaces have been available for C and C++ developers only. Now, Add-in Express hides XLL complexities for .NET developers.

To create an XLL add-in, choose *Add-in Express XLL Add-in* in the New Project dialog. The project designer class is ADXXLLModule (XLL add-in module). The module contains a special class, XLLContainer, where you add your `public static` (in VB, `Public Shared`) functions. Just adding a function is enough for a quick start. Using the module's designer, you are able to specify all other function-related stuff: description, help reference, category, descriptions of the function's parameters, etc. In addition, you can instruct Excel to call your function whenever recalculation is required (IsVolatile property). Another option is specifying a parameter of the Object type to accept Excel ranges as a reference to an object of the ADXExcelRef type or as a 2D array of values.

> Multi-threaded calculations introduced in Excel 2007 are not supported.

What's next?

Two sample projects are described in Your First Excel Automation Add-in and Your First XLL add-in. Also, find plenty of useful information in Excel UDFs.

Excel Workbooks

Sometimes you need to automate a given Excel workbook (template). You can do it with ADXExcelSheetModule that represents one worksheet of the workbook. The Document property allows creating and browsing for the workbook. If you choose creating a new workbook, the dialog appears where you specify the name and location of the workbook as well as the Property Name and Property Value textboxes. Add-in Express adds this property to the list of custom properties of the workbook and uses the name and value of the property in order to recognize the workbook. Accordingly, you specify the PropertyId

and PropertyValue properties of the module. The module provides a full set of events available for an Excel workbook.

For the Add-in Express components available for the module see the following chapters: Command Bar UI, Connecting to Existing CommandBar Controls and Application-level Events.

There is a sample project for this module type. It is called TimeSheet. Together with other sample projects, it can be downloaded at http://www.add-in-express.com/downloads/adxnet.php.

Word Documents

To automate a given Word document, you use ADXWordDocumentModule. This module allows creating and browsing for the document. If you choose creating a new document, the dialog appears where you specify the name and location of the document as well as the Property Name and Property Value textboxes. Add-in Express adds this property to the list of custom properties of the document and uses the name and value of the property in order to recognize the document. Accordingly, you specify the PropertyId and PropertyValue properties of the module. The module provides a full set of events available for a Word document.

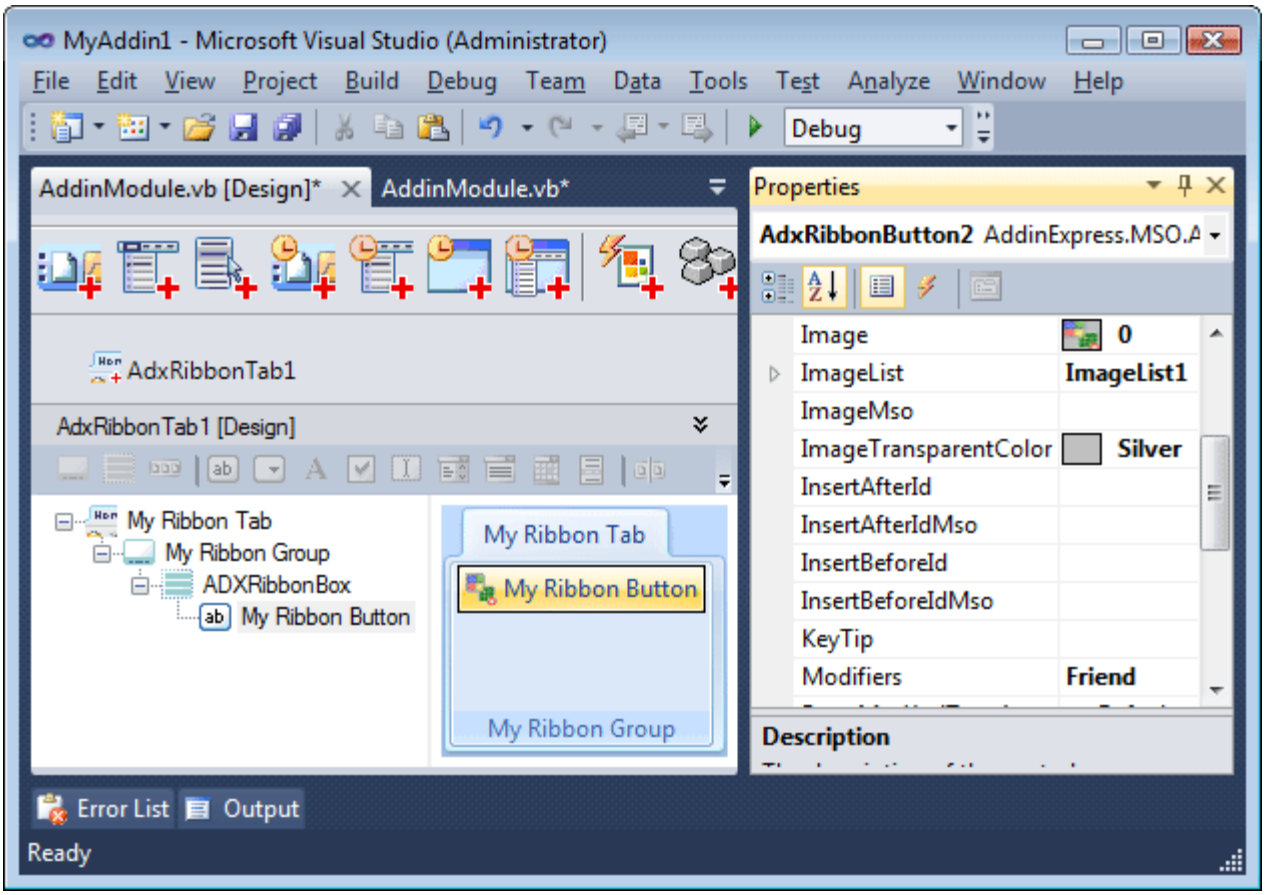
For the Add-in Express components available for the module see the following chapters: Command Bar UI, Connecting to Existing CommandBar Controls and Application-level Events.

There is a sample project for this module type. It is called WordFax. Together with other sample projects, it can be downloaded at http://www.add-in-express.com/downloads/adxnet.php.

Add-in Express Components

Ribbon UI

Office 2007 presented a new Ribbon user interface. Microsoft states that the interface makes it easier and quicker for users to achieve the wanted results. The developers extend this interface by using the XML markup that the COM add-in should return to the host through an appropriate interface when your add-in is loaded into the host application version supporting the Ribbon UI.



Add-in Express provides some 50 Ribbon components that undertake the task of creating the markup. Also, there are 5 visual designers that allow creating the Ribbon UI of your add-in: Ribbon Tab (ADXRibbonTab), Ribbon Office Menu (ADXRibbonOfficeMenu), Quick Access Toolbar (ADXRibbonQuickAccessToolbar), Ribbon BackstageView (ADXBackStageView), and Ribbon Context Menu (ADXRibbonContextMenu).

In Office 2010, Microsoft abandoned the Office Button (introduced in Office 2007) in favor of the *File Tab* (Backstage View). To provide some sort of compatibility for you, ADXRibbonOfficeMenu will map your controls to the File tab **unless** you use ADXBackStageView components in your project; otherwise, all the controls you add to ADXRibbonOfficeMenu are ignored when Office 2010 loads your add-in.

Microsoft require developers to use the StartFromScratch parameter (see the StartFromScratch property of the add-in module) when customizing the Quick Access Toolbar.

See also Your First Microsoft Office COM Add-in, Your First Microsoft Outlook COM Add-in.

How Ribbon Controls Are Created?

When your add-in is being loaded by the host application supporting the Ribbon UI, the very first event received by the add-in is the OnRibbonBeforeCreate event of the add-in module. This is the only event in which you can add/remove/modify the Ribbon components onto/from/on the add-in module.

Then Add-in Express generates the XML markup reflecting the settings of the Ribbon components and raises the OnRibbonBeforeLoad event. In that event, you can modify the generated markup, say, by adding XML tags generating extra Ribbon controls.

Finally, the markup is passed to Office and the add-in module fires the OnRibbonLoaded event. In the event parameters, you get an object of the AddinExpress.MSO.IRibbonUI type that allows invalidating a Ribbon control; you call the corresponding methods when you need the Ribbon to re-draw the control. Also, in Office 2010 only, you can call a method activating a Ribbon tab.

Remember, the Ribbon designers perform the XML-schema validation automatically, so from time to time you may run into the situation when you cannot add a control to some level. It is a restriction of the Ribbon XML-schema.

Still, we recommend turning on the Ribbon XML validation mechanism through the UI of the host application of your add-in; you need to look for a checkbox named *"Show add-in user interface errors"*.

Referring to Built-in Ribbon Controls

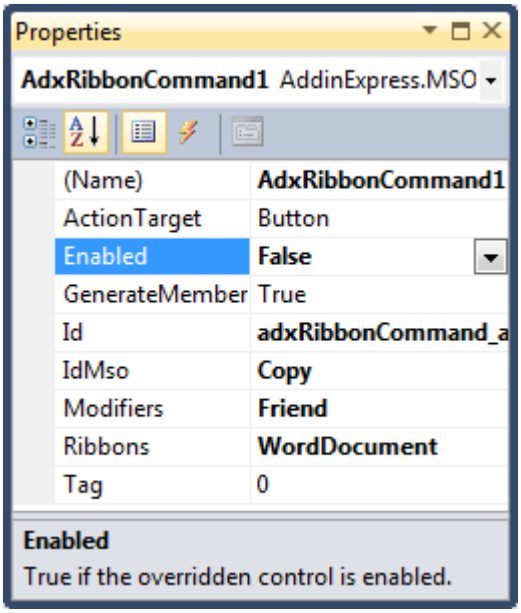
All built-in Ribbon controls are identified by their IDs. Pay attention, **the ID of a built-in Ribbon control is a string**, not integer. All such IDs are available for download on the Microsoft web site, for Office 2007, see here; for Office 2010, see this page. The download installs Excel files; the Control Name column of each contains the IDs of **almost** all built-in Ribbon controls for the corresponding Ribbon.

Add-in Express Ribbon components provide the IdMso property; if you leave it empty the component will create a custom Ribbon control. To refer to a built-in Ribbon control, you set the IdMso property of the component to the ID of the built-in Ribbon control. For instance, you can add a custom Ribbon group to a built-in tab. To do this, you add a Ribbon tab component onto the add-in module and set its IdMso to the ID of the required built-in Ribbon tab. Then you add your custom group to the tab and populate it with controls. Note that the Ribbon does not allow adding a custom control to a built-in Ribbon group.

Intercepting Built-in Ribbon Controls

You use the *Ribbon Command* (ADXRibbonCommand) component to override the default action of a built-in Ribbon control. Note that the Ribbon allows intercepting only buttons, toggle buttons and check boxes; see the ActionTarget property of the component. You specify the built-in Ribbon control to be intercepted in the IdMso property of the component. In fact, you are supposed to specify the ID of the control to be intercepted. To get all such IDs, see Referring to Built-in Ribbon Controls.

Another use of the component is shown in the screenshot below; the following settings disable the *Copy* command in Word 2007-2010:



Positioning Ribbon Controls

Every Ribbon component provides the InsertBeforeId, InsertBeforeIdMso and InsertAfterId, InsertAfterIdMso properties. You use the InsertBeforeId and InsertAfterId properties to position the control among other controls created by your add-in, just specify the Id of the corresponding Ribbon components in any of these properties. The InsertBeforeIdMso and InsertAfterIdMso properties allow positioning the control among built-in Ribbon controls (see also Referring to Built-in Ribbon Controls).

Creating Ribbon Controls at Run-time

You cannot create Ribbon controls at run-time (but see How Ribbon Controls Are Created?) because Ribbon is a static thing from birth; the only control providing any dynamism is Dynamic Menu (see the Dynamic property of the ADXRibbonMenu component). For other control types, you can only imitate that dynamism by changing the Visible property of a Ribbon control.

Properties and Events of the Ribbon Components

Add-in Express Ribbon components implement two schemas of refreshing Ribbon controls.

The simple schema allows you to change a property of the Ribbon component and the component will supply it to the Ribbon UI whenever it requests that property. This mechanism is an ideal when you need to display static or almost static things such as a button caption that doesn't change or changes across all windows showing the button, say in Outlook inspectors or Word documents. This works because Add-in Express supplies the same value for the property whenever the Ribbon UI invokes a corresponding callback function.

However, if you need to have a full control over the Ribbon UI, say, when you need to show different captions of a Ribbon button in different Inspector windows or Word documents, you can use the OnPropertyChanging event provided by all Ribbon components. That event occurs when the Ribbon expects that you can supply a new value for a property of the Ribbon control. The event allows you to learn the current context, i.e. the current window showing your Ribbon controls, such as Outlook.Inspector, Word.Document, etc. It also allows you to get the property being changed and its current value. Finally, you can change that value as required.

Sharing Ribbon Controls Across Multiple Add-ins

First off, you assign the same string value to the AddinModule.Namespace property of every add-in that will share your Ribbon controls. This makes Add-in Express add two xmlns attributes to the customUI tag in the resulting XML markup:

- xmlns:default="%ProgId of your add-in, see the ProgID attribute of the AddinModule class%",
- xmlns:shared="%the value of the AddinModule.Namespace property%".

Originally, all Ribbon controls are located in the default namespace (id="%Ribbon control's id%" or idQ="default:%Ribbon control's id%") and you have full control over them via the callbacks provided by Add-in Express. When you specify the Namespace property, Add-in Express changes the markup to use idQ's instead of id's.

Then, in all add-ins that are to share a Ribbon control, for the control with the same Id (you can change the Id's to match), you set the Shared property to true. For the Ribbon control whose Shared property is true, Add-in Express changes its idQ to use the shared namespace (idQ="shared:%Ribbon control's id%") instead of the default one. Also, for such Ribbon controls, Add-in Express cuts out all callbacks and replaces them with "static" versions of the attributes. Say, getVisible="getVisible_CallBack" will be replaced with visible="%value%".

The shareable Ribbon controls are the following Ribbon container controls:

- Ribbon Tab - ADXRibbonTab

- Ribbon Box - ADXRibbonBox
- Ribbon Group - ADXRibbonGroup
- Ribbon Button Group - ADXRibbonButtonGroup

When referring to a shared Ribbon control in the BeforeId and AfterId properties of another Ribbon control, you use the shared controls' idQ: %namespace abbreviation% + ":" + %control id%. The abbreviations of these namespaces are "default" and "shared" string values.

Say, when creating a shared tab, containing a private group, containing a button (private again), the resulting XML markup looks as follows:

```
<customUI xmlns="http://schemas.microsoft.com/office/2006/01/customui"
          xmlns:default="MyOutlookAddin1.AddinModule"
          xmlns:shared="MyNameSpace" [callbacks omitted]>
 <ribbon>
  <tabs>
   <tab idQ=" shared:adxRibbonTab1" visible="true" label="My Tab">
    <group idQ="default:adxRibbonGroup1" [callbacks omitted]>
     <button idQ="default:adxRibbonButton1" [callbacks omitted]/>
    </group>
   </tab>
  </tabs>
 </ribbon>
</customUI>
```

Task Panes

Custom Task Panes in Office 2007-2010

To allow further customization of Office applications, they introduced custom task panes in Office 2007. Add-in Express supports custom task panes by equipping the COM add-in module with the TaskPanes property. Add a UserControl to your project, add an item to the TaskPanes collection of the add-in module, and set up the item by choosing the control in the ControlProgId property and filling in the Title property. Add your reaction to the OnTaskPaneXXX event series of the add-in module and the DockPositionStateChange and VisibleStateChange events of the task pane item. Use the OfficeColorSchemeChanged event and the OfficeColorScheme property to get the current Office color scheme. See a sample in Custom Task Panes (Office 2007+).

Advanced Custom Task Panes in Office 2000-2010

Add-in Express allows showing advanced custom task panes in Outlook, Excel, Word and PowerPoint, versions 2000-2010. See Advanced Custom Task Panes for details.

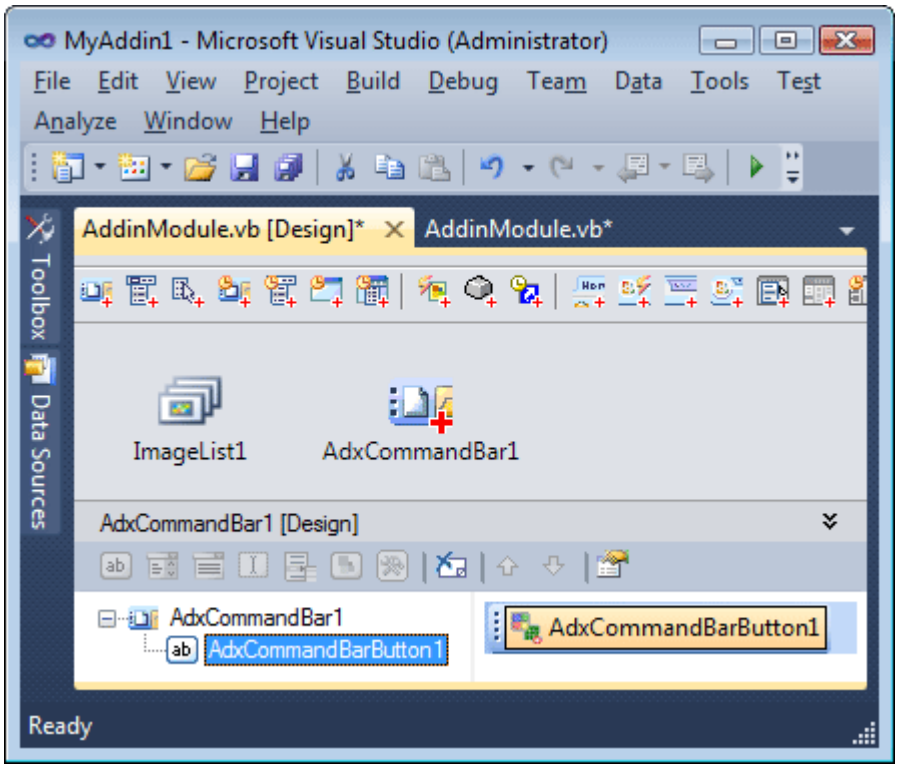
Command Bar UI

This section describes components for creating the UI of your add-in in Office 2000-2003 and in non-Ribboned applications of Office 2007: Outlook 2007 (Explorer windows only), Publisher 2007, Visio 2007, Project 2007, InfoPath 2007.

In all other applications, the command bar UI has been superseded by the new Ribbon user interface. Nevertheless, all command bars and controls are still available in those Office applications and you may want to use this fact in your code. Also, custom command bar controls created by your add-in will be shown on the *Add-ins* tab in the Ribbon UI but the best way is to support both CommandBar and Ribbon user interfaces in your add-in, To do this, you need to add both command bar and ribbon components onto the add-in module.

The command bar UI of your add-in includes custom and built-in command bars as well as custom and built-in command bar controls. Command bar is a common term for traditional toolbars, menus, and context menus.

Add-in Express provides toolbar, main menu, and context menu components that allow tuning up targeted command bars at design-time. There are also Outlook-specific versions of toolbar and main menu components. Every such component provides an in-place visual designer. For instance, the screenshot below shows a visual designer for the toolbar component that creates a custom toolbar with a button.



To create toolbars and menus in Outlook, you need to use Outlook-specific versions of command bar components. See Outlook Toolbars and Main Menus.

Using visual designers, you populate your command bars with controls and set up their properties at design-time. At run-time, you use the Controls collection provided by every command bar component. Every control (built-in and custom) added to this collection will be added to the corresponding toolbar at your add-in startup. See also How Command Bars and Their Controls Are Created and Removed?

Toolbar

To add a toolbar to the host application, use the *Add ADXCommandBar* command available in the *Commands* toolbar of the add-in module designer. It adds an ADXCommandBar component to the module. The most important property of the component is CommandBarName. If its value is not equal to the name of any built-in command bar of the host application, then you are creating a new command bar. If its value is equal to any built-in command bar of the host application, then you are connecting to a built-in command bar. To find out the built-in command bar names, use our free Built-in Controls Scanner utility.

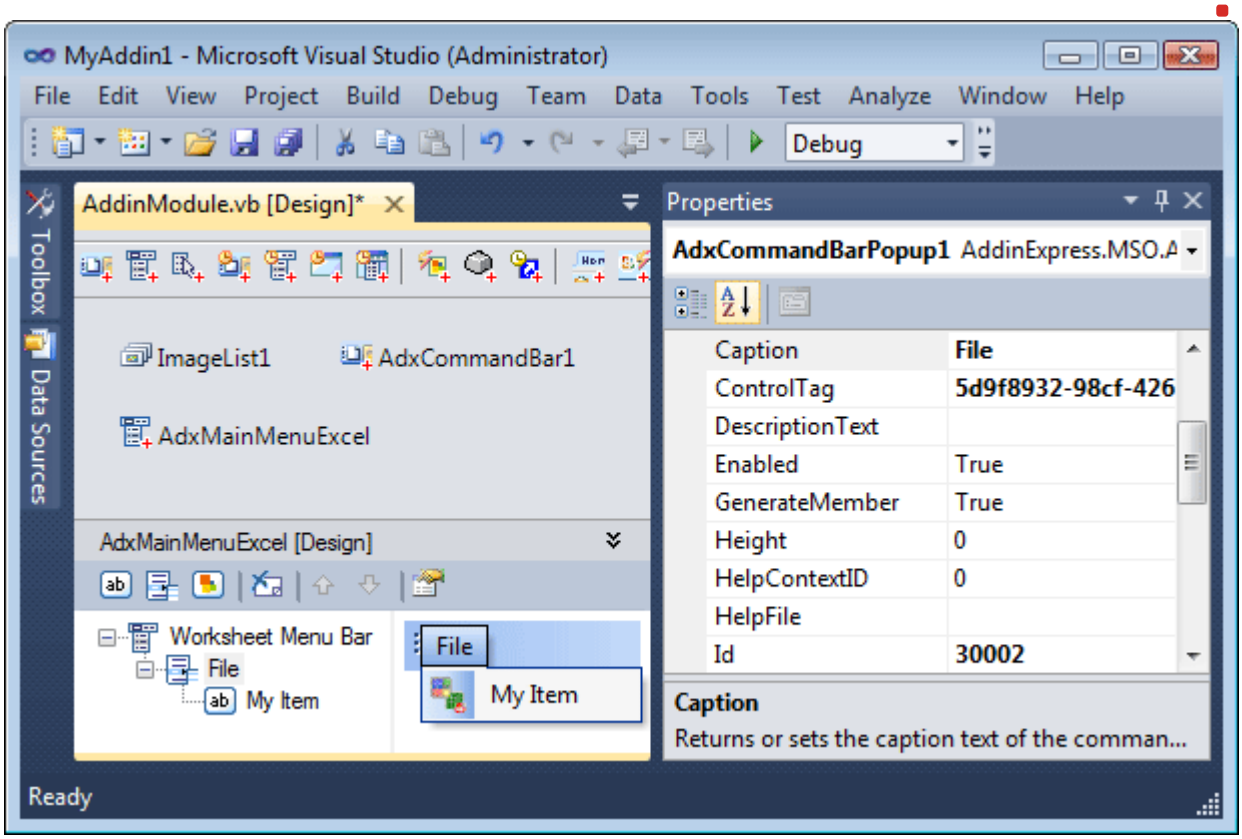
To position a toolbar, use the Position property that allows docking the toolbar to the top, right, bottom, or left edges of the host application window. You can also leave your toolbar floating. For a fine positioning, you use the Left, Top, and RowIndex properties. To show a pre-2007 toolbar in the *Add-ins* tab in Office 2007-2010, set the UseForRibbon property of the corresponding command bar component to true.

Pay attention to the SupportedApps property. You use it to specify if the command bar will appear in some or all host applications supported by the add-in. Using several command bar components with different values in their SupportedApps properties is useful when creating toolbars for Outlook and Word (see below). Unregister your add-in before you change the value of the SupportedApps property.

To speed up add-in loading when connecting to an existing command bar, set the Temporary property to False. To make the host application remove the command bar when the host application quits, set the Temporary property to True. However, this is the general rule only. If your add-in supports Outlook or Word, see How Command Bars and Their Controls Are Created and Removed? You need to unregister the add-in before changing the value of this property.

Main Menu

By using the Add Main Menu command of the add-in module (see Commands of the Add-in Module), you add an ADXMainMenu, which is intended for customizing the main menu in an Office application, which you specify in the SupportedApp property.



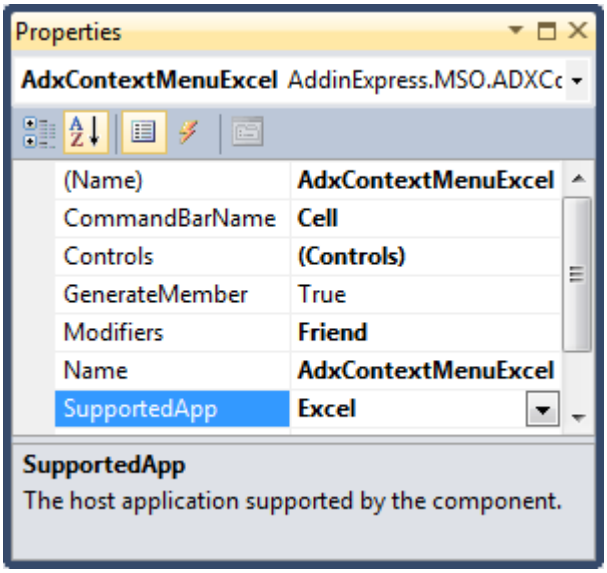
Like the toolbar component, it provides a visual designer for the Controls property. To add a custom top-level menu item, just add a popup control to the command bar. Then you can populate it with other controls. Note, however, that for all menu components, controls can be buttons and pop-ups only. To add a custom button to a built-in top-level menu item, you specify the ID of the top-level menu item in the Id property of the button control. For instance, the ID of the File menu item in all Office applications is 30002. See more details about IDs of command bar controls in Connecting to Existing CommandBar Controls. In main applications of Office 2007, they replaced the command system with the Ribbon UI. Therefore, instead of adding custom items to the main menu, you need to add them to a custom or built-in Ribbon tab. Also, you can add custom items to the menu of the *Office Button* in Office 2007. In Office 2010, they abandoned the Office button in favor of the *File Tab*, also known as Backstage View. Add-in Express provides components allowing customizing both the *File Tab* and the Ribbon *Office Menu*, see Step #11 – Customizing the Ribbon User Interface in Your First Microsoft Office COM Add-in. Note, if you customize the Office Button menu only, Add-in Express will map your controls to the Backstage View when the add-in is run in Office 2010. If, however, both Office Button menu and File tab are customized at the same time, Add-in Express ignores custom controls you add to the Office Button menu.

Context Menu

In Office 2000-2007, context menus are command bars and they can be customized in the same way as any other command bar. In Office 2010, they allow us to customize context menus via the Ribbon XML. Accordingly, Add-in Express provides two components: a commandbar-based (ADXContextMenu) and Ribbon-based (ADXRibbonContextMenu).

The PowerPoint development team explicitly states that PowerPoint 2007 doesn't support customizing context menus with command bar controls. However, some context menus in PowerPoint 2007 are still customizable in this way.

The *Add ADXContextMenu* command of the add-in module adds an ADXContextMenu, which allows adding a custom command bar control to any context menu available in all Office 2000-2007 applications **except for Outlook 2000**. The component allows connecting to a single context menu of a single host application. Like for the ADXMainMenu component, you must specify the SupportedApp property. To specify the context menu you want to connect to, just choose the name of the context menu in the CommandBarName combo.



Please note that the context menu names for this property were taken from Office 2007, the last Office version that introduced **new** commandbar-based context menus. That is, it is possible that the targeted context menu is not available in a pre-2007 Office version.

In Office 2010 and higher, you can customize both commandbar-based and Ribbon-based context menus. See Step #8 – Customizing Context Menus and Step #9 – Customizing Context Menus in Outlook.

Outlook Toolbars and Main Menus

While the look-n-feel of all Office toolbars is the same, Outlook toolbars differ from toolbars of other Office applications. They are different for the two main Outlook window types – for Outlook Explorer and Outlook Inspector windows. Accordingly, Add-in Express provides you with Outlook-specific command bar components that work correctly in multiple Explorer and Inspector windows scenarios: ADXOlExplorerCommandBar and ADXOlInspectorCommandBar. In the same way, Add-in Express

provides Outlook-specific versions of the Main Menu component: ADXOlExplorerMainMenu and ADXOlInspectorMainMenu. See Commands of the Add-in Module.

All of the components above provide the FolderName, FolderNames, and ItemTypes properties that add context-sensitive features to the command bar. For instance, you can choose your toolbar to show up for e-mails only. To get this, just check the correct checkbox in the ItemTypes property editor.

Connecting to Existing Command Bars

In Office, all command bars are identified by their names. Specifying the name of a toolbar in the ADXCommandBar.CommandBarName property means referring to that toolbar. Use our free Built-in Controls Scanner to get the names of all built-in command bars in any Office 2000-2010 application.

Connecting to Existing CommandBar Controls

Any *CommandBar Control* component connects to **a built-in control** using the Id property. That is, if you set the Id property of the component to an integer other than 1 and a built-in control having the same ID exists on the specified command bar, the component connects to the built-in control and ignores all other properties. If no such control is found, the component adds it to the command bar.

Using the approach below, you can override the standard behavior of a built-in button on a given toolbar:

- Add a new toolbar component to the module
- Specify the toolbar name in the CommandBarName property
- Add an ADXCommandBarButton to the command bar
- Specify the ID of the built-in button in the ADXCommandBarButton.Id property
- Set ADXCommandBarButton.DisableStandardAction to true
- Now you should handle the Click event of the button

Also, you can use the Built-in Control Connector component, which allows overriding the standard action for any built-in control (without adding it onto any command bar):

- Add a built-in control connector onto the module.
- Set its Id property to the ID of your command bar control.
- To connect the component to all instances of the command bar control having this ID, leave its CommandBar property empty. To connect the component to the control on a given toolbar, specify the toolbar in the CommandBar property.
- To override and/or cancel the default action of the control, use the ActionEx event.

The component traces the context and when any change happens, it reconnects to the currently active instance of the command bar control with the given Id, taking this task away from you.

You can find the IDs of built-in command bar controls using the free Built-in Controls Scanner utility. Download it at http://www.add-in-express.com/downloads/controls-scanner.php.

How Command Bars and Their Controls Are Created and Removed?

When your add-in is being loaded by the host application, the add-in module raises the AddinInitialize event before processing command bar components. In most Office applications except for Outlook, this is the last event in which you may add/remove/modify command bar components onto/from/on the add-in module. For instance, you can delete some or all of the command bar components if the environment in which your add-in is being loaded doesn't meet some requirements. After that event, Add-in Express scans components on the add-in module, creates new or connects to existing toolbars and raises the AddinStartupComplete event.

All command bar and commandbar control components provide the Temporary property of the Boolean type. Temporary toolbars and controls are not saved when the host application quits. This causes the creation of such toolbars and controls at every add-in startup. Permanent toolbars and controls are saved by the host application and restored at startup; i.e. permanent toolbars allow your add-in to load faster. But Word and Outlook require specific approaches to temporary/permanent toolbars and controls.

Let's look at how command bars and controls are removed, however. When the user turns the add-in off in the COM Add-ins Dialog, Add-in Express uses a method of the IDTExtenisbility2 interface to remove the command bars and controls. When the add-in is uninstalled, and there are non-temporary toolbars and controls in the add-in, Add-in Express starts the host application and removes the toolbars and controls. That is, temporary toolbars and controls allow your add-in to uninstall faster.

Let's get back to Outlook and Word, however.

It is strongly recommended that you use temporary command bars and controls in Outlook add-ins. If they are non-temporary, Add-in Express will run Outlook to remove the command bars when you uninstall the add-in. Now imagine Outlook asking the user to select a profile or enter a password...

In Word add-ins, we strongly advise making **both** command bars and controls non-temporary. Word removes temporary command bars. However, it doesn't remove temporary command bar controls, at least some of them; it just hides them. When the add-in starts for the second time, Add-in Express finds such controls and connects to them. Accordingly, because Add-in Express doesn't change the visibility of existing controls, the controls are missing in the UI.

Note that main and context menus are command bars. That is, in Word add-ins, custom controls added to these components must have Temporary = False, too. If you set Temporary = True for such controls (say, by accident), they will not be removed when you uninstall your add-in. That happens because Word has another peculiarity: it saves temporary controls when they are added to a built-in command bar. And all

context menus are built-in command bars. To remove such controls, you will have to write some code or use a simple way: set Temporary to false for all controls, register the add-in on the affected PC, run Word. At this moment, the add-in finds this control and traces it from this moment on. Accordingly, when you unregister the add-in, the control is removed in a standard way.

Several notes.

When debugging your add-in, you need to unregister it before changing the Temporary property. After changing the property, register the add-in anew.

For every permanent toolbar (ADXCommandBar.Temporary = False), Add-in Express creates a registry key in `{HKLM or HKCU}\Software\Microsoft\Office\{host application}\Addins\{your add-in}\Commandbars` when the host application quits. The key is used to detect a scenario in which the user removes the toolbar form the UI: if both the key and the toolbar are missing, Add-in Express creates the toolbar. You may need to use this fact in some situations.

Command Bars in the Ribbon UI

By default, Add-in Express doesn't show custom command bar controls or main menu items when your add-in is loaded by a Ribbon-enabled application. This behavior is controlled by the UseForRibbon property of the ADXCommandBar, ADXOlExplorerCommandBar, ADXOlInspectorCommandBar, ADXMainMenu, ADXOlExplorerMainMenu, or ADXOlInspectorMainMenu components. If you set it to True, the Ribbon places corresponding controls on the *Add-ins* tab in the Ribbon UI.

Usually, you set that property at design-time. You can also set this property at run-time but this must be done before Add-in Express processes the corresponding component to create a command bar and its controls. The best moment for doing this is the AddinInitialize event of ADXAddinModule.

As to the context menus, Ribbon-enabled applications of the Office 2007 suite demonstrate lack of coordination: most of them support customizing their context menus with command bar controls (remember, in Office 2007, context menus are still command bars) but the PowerPoint development team explicitly states that PowerPoint 2007 doesn't support this. Note that Office 2010 provides support for both commandbar-based and Ribbon-based context menus; see Step #8 – Customizing Context Menus and Step #9 – Customizing Context Menus in Outlook

Command Bar Control Properties and Events

The main property of any command bar control (they descend from ADXCommandBarControl) is the Id property. A custom command bar control has ID = 1; all built-in controls have IDs of their own. To add a custom control to the toolbar, leave the Id unchanged. To add a built-in control to your toolbar, specify its ID in the corresponding property of the command bar control component. To find out the ID of every built-in control in any Office application, use our free Built-in Controls Scanner utility.

To add a separator before any given control, set its BeginGroup property to true.

Set up a control's appearance using a large number of its properties, such as Enabled and Visible, Style and State, Caption and ToolTipText, DropDownLines and DropDownWidth, etc. You also control the size (Height, Width) and location (Before, AfterId, and BeforeId) properties. To provide your command bar buttons with a default list of icons, drop an ImageList component onto the add-in module and specify the ImageList in the Images property of the module. Do not forget to set the button's Style property to either adxMsoButtonIconAndCaption or adxMsoButtonIcon. See also Transparent Icon on a CommandBarButton.

Use the OlExplorerItemTypes, OlInspectorItemTypes, and OlItemTypesAction properties to add context-sensitivity to your controls on Outlook-specific command bars. The OlItemTypesAction property specifies an action that Add-in Express will perform with the control when the current item's type coincides with that specified by you.

To handle user actions, use the Click event for buttons and the Change event for edit, combo box, and drop down list controls. Use also the DisableStandardAction property available for built-in buttons added to your command bar. To intercept events of any built-in control, see Connecting to Existing CommandBar Controls.

Command Bar Control Types

The Office Object Model contains the following control types available for **toolbars**: button, combo box, and pop-up. Using the correct property settings of the combo box component, you can extend the list with edits and dropdowns.

Nevertheless, this list is extremely short. Add-in Express allows extending this list with any .NET control (see Toolbar Controls for Microsoft Office). You can add controls using that technology onto old-fashioned toolbars; that possibility is not available for Office applications showing the Ribbon UI.

Please note that due to the nature of command bars, **menu and context menu items** can only be buttons and pop-ups (item File in any main menu is a sample of a popup).

Outlook UI Components

Outlook Bar Shortcut Manager

Outlook provides us with the Outlook Bar (Navigation Pane in Outlook 2003-2010). The Outlook Bar displays Shortcut groups consisting of Shortcuts that you can target a Microsoft Outlook folder, a file-system folder, or a file-system path or URL. You use the Outlook Bar Shortcut Manager to customize the Outlook Bar with your shortcuts and groups.

This component is available for ADXAddinModule. Use the Groups collection of the component to create a new shortcut group. Use the Shortcuts collection of a short group to create a new shortcut. To connect to an

existing shortcut or shortcut group, set the Caption properties of the corresponding ADXOlBarShortcut and/or ADXOlBarGroup components equal to the caption of the existing shortcut or shortcut group. Please note that there is no other way to identify the group or shortcut.

That is why your shortcuts and shortcut groups must be named uniquely for Add-in Express to remove only the specified ones (and not those having the same names) when the add-in is uninstalled. If you have several groups (or shortcuts) with the same name, you will have to remove them yourself. Depending on the type of its value, the Target property of the ADXOlBarShortcut component allows you to specify different shortcut types. If the type is Outlook.MAPIFolder, the shortcut represents a Microsoft Outlook folder. If the type is String, the shortcut represents a file-system path or a URL. No events are available for these components.

Outlook Property Page

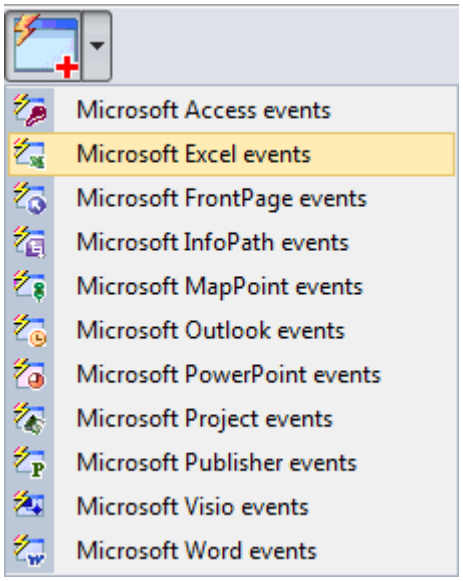
Outlook allows extending its Options dialog with custom pages. You see this dialog when you choose *Tools | Options* menu. In addition, Outlook allows adding such page to the Folder Properties dialog. You see this dialog when you choose the Properties item in the folder context menu. You create such pages using the Outlook Property Page component.

In the Add New Item dialog, choose the Outlook Options Page item to add a class to your project. This class is a descendant of the System.Windows.Forms.UserControl class. It allows creating Outlook property pages using its visual designer. Just set up the property page properties, place your controls onto the page, and add your code. To add this page to the Outlook Options dialog, select the name of your control class in the PageType combo of ADXAddinModule and enter some characters into the PageTitle property.

To add a page to the *Folder Properties* dialog for a given folder(s), you use the FolderPages collection of the add-in module. Run its property editor and add an item (of the ADXOlFolderPage type). You connect the item to a given property page through the PageType property. Note, the FolderName, FolderNames, and ItemTypes properties of the ADXOlFolderPage component work in the same way as those of Outlook-specific command-bars.

Specify reactions required by your business logics in the Apply and Dirty event handlers. Use the OnStatusChange method to raise the Dirty event, the parameters of which allow marking the page as Dirty.

Events



Application-level Events

ADXAddinModule provides events for all Office applications through the *Add Events* command that adds and/or removes appropriate Add-in Express event components to the module. Use the event handlers of an Add-in Express event component to respond to the host application's events. You may need to process other events provided by Outlook and Excel. If this is the case, see Events Classes.

Events Classes

Outlook and Excel differ from other Office applications because they have event-raising objects not only at the topmost level of their object models. These exceptions are the Folders and Items classes as well as all item types (MailItem, TaskItem etc.) in Outlook, and the Worksheet class in Excel. Add-in Express events classes provide you with version independent components that ease the pain of handling such events. The events classes also handle releasing of COM objects required for their functioning.

At design-time, you add an events class to the project (see Creating Add-in Express Projects) and use its event procedures to write the code for just one set of event handling rules for a given event source type, say for, the Items collection of the MAPIFolder class in Outlook 2000-2003; in Outlook 2007-2010, you can also use the Folder class. To implement another set of event handling rules for the same event source type, you add another events class to your project.

At run-time, you connect an events class to an event source using the ConnectTo method. To disconnect the events class from the event source you use the RemoveConnection method. To apply the same business rules to another event source of the same type (say, to items of another folder), you create a new instance of the same events class.

What follows below is the source code of a newly added events class that processes the events of the Items collection of the MAPIFolder class in Outlook (Folder class in Outlook 2007).

```
Imports System

'Add-in Express Outlook Items Events Class
Public Class OutlookItemsEventsClass1
   Inherits AddinExpress.MSO.ADXOutlookItemsEvents

   Public Sub New(ByVal ADXModule As AddinExpress.MSO.ADXAddinModule)
      MyBase.New(ADXModule)
```

```
   End Sub

   Public Overrides Sub ProcessItemAdd(ByVal Item As Object)
      'TODO: Add some code
   End Sub

   Public Overrides Sub ProcessItemChange(ByVal Item As Object)
      'TODO: Add some code
   End Sub

   Public Overrides Sub ProcessItemRemove()
      'TODO: Add some code
   End Sub
End Class
```

Intercepting Keyboard Shortcuts

Every Office application provides built-in keyboard combinations that allow shortening the access path for commands, features, and options of the application. Add-in Express allows adding custom keyboard combinations and processing both custom and built-in ones.

Add a Keyboard Shortcut component onto the add-in module, choose or specify the keyboard shortcut you need in the ShortcutText property, set the HandleShortCuts property of the module to true and process the Action event of the component.

Smart Tag

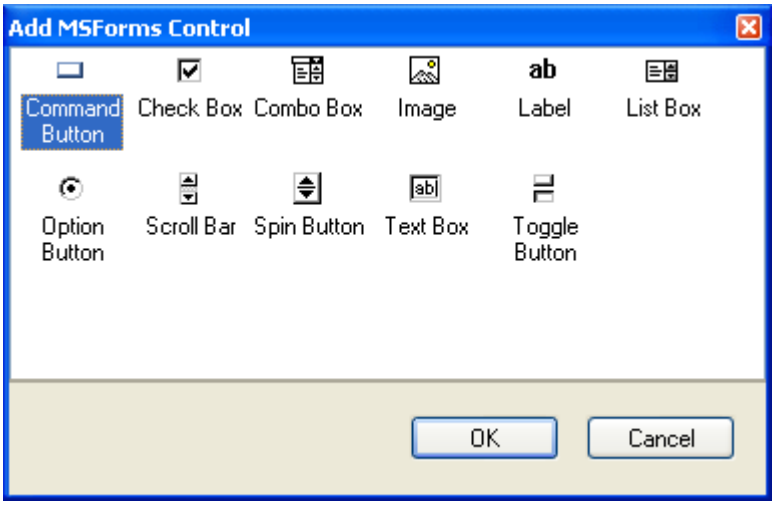
The Kind property of the ADXSmartTag component allows you to choose one of two text recognition strategies: either using a list of words in the RecognizedWords string collection or implementing a custom recognition process based on the Recognize event of the component. Use the ActionNeeded event to change the Actions collection according to the current context. The component raises the PropertyPage event when the user clicks the Property button in the Smart Tags tab (Tools / AutoCorrect Options menu) for your smart tag.

RTD Topic

Use the String## properties to identify the topic of your RTD server. To handle startup situations nicely, specify the default value for the topic and, using the UseStoredValue property, specify, if the RTD function in Excel returns the default value (UseStoredValue = false) or doesn't change the displayed value (UseStoredValue = true). The RTD topic component provides you with the Connect, Disconnect, and RefreshData events. The last one occurs (for enabled topics only) whenever Excel calls the RTD function.

MSForms Control

This command is available for ADXExcelSheetModule and ADXWordDocumentModule. When run, it displays the following dialog:



Select the control you need to connect to and click OK. Add-in Express adds an appropriate MS Forms Control Connector to the module. Use the ControlName property of the connector to specify the underlying control on the Excel worksheet or Word document. Respond to the events provided by the control connector.

Advanced Custom Task Panes

Add-in Express allows COM add-ins to show custom panes in Outlook, Excel, Word, and PowerPoint, versions 2000-2010.

An Absolute Must-Know

Here are the three main points you should be aware of:

- there are application-specific <Manager> components; every <Manager> component provides a collection; each <Item> from the collection binds a <Form> (an application-specific descendant of System.Windows.Forms.Form) to the visualization and context (Outlook-only) settings;
- you **never** create an instance of a <Form> in the way you create an instance of System.Windows.Forms.Form; instead, the <Manager> creates instances of the <Form> for you; the instances are created either automatically or at your request;
- the Visible property of a <Form> instance is true, when the instance is embedded into a window region (as specified by the visualization settings) regardless of the actual visibility of the instance; the Active property of the <Form> instance is true, when the instance is shown on top of all other instances in the same region.

A required comment

Anywhere in this section, a term in angle brackets, such as <Manager> or <Form> above, specifies a component, class, or class member, the actual name of which is application-dependent. Every such term is covered in the corresponding chapter of this manual.

Hello, World!

Adding custom panes in a particular application is described in appropriate parts of the following samples:

- Outlook – in Your First Microsoft Outlook COM Add-in see Step #10 – Adding a Custom Task Pane in Outlook 2000-2010
- Excel – in Your First Microsoft Office COM Add-in, see Step #12 – Adding Custom Task Panes in Excel 2000-2010
- PowerPoint - in Your First Microsoft Office COM Add-in, see Step #13 – Adding Custom Task Panes in PowerPoint 2000-2010

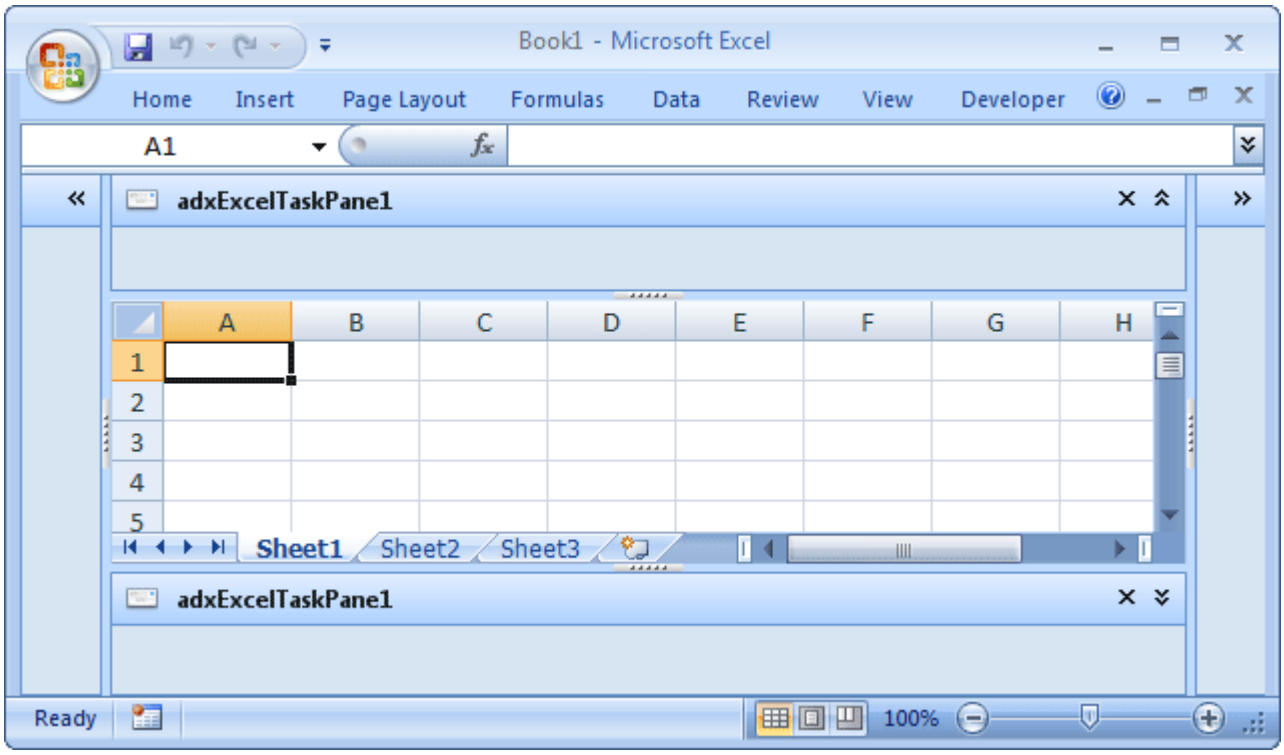
- Word – in Your First Microsoft Office COM Add-in, see Step #14 – Adding Custom Task Panes in Word 2000-2010

The Regions

Obviously, all Office applications have different window structures. Accordingly, Add-in Express provides a number of application-specific options for embedding your forms.

Word, Excel and PowerPoint Regions

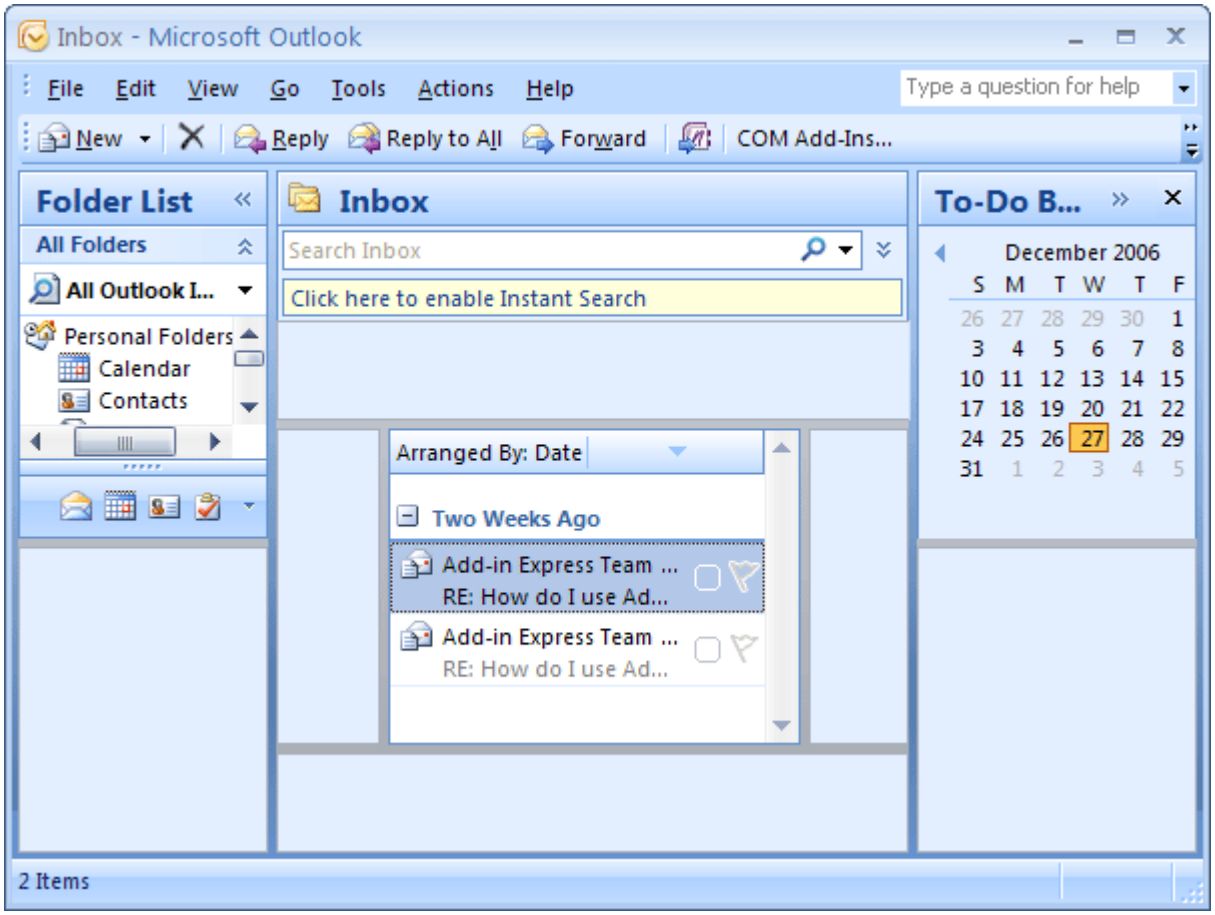
These Office applications allow showing your forms in four regions; the regions are docked to the four edges of the application's main window. The names of the regions are Left, Top, Right, and Bottom (see the Position property of the <Item>).



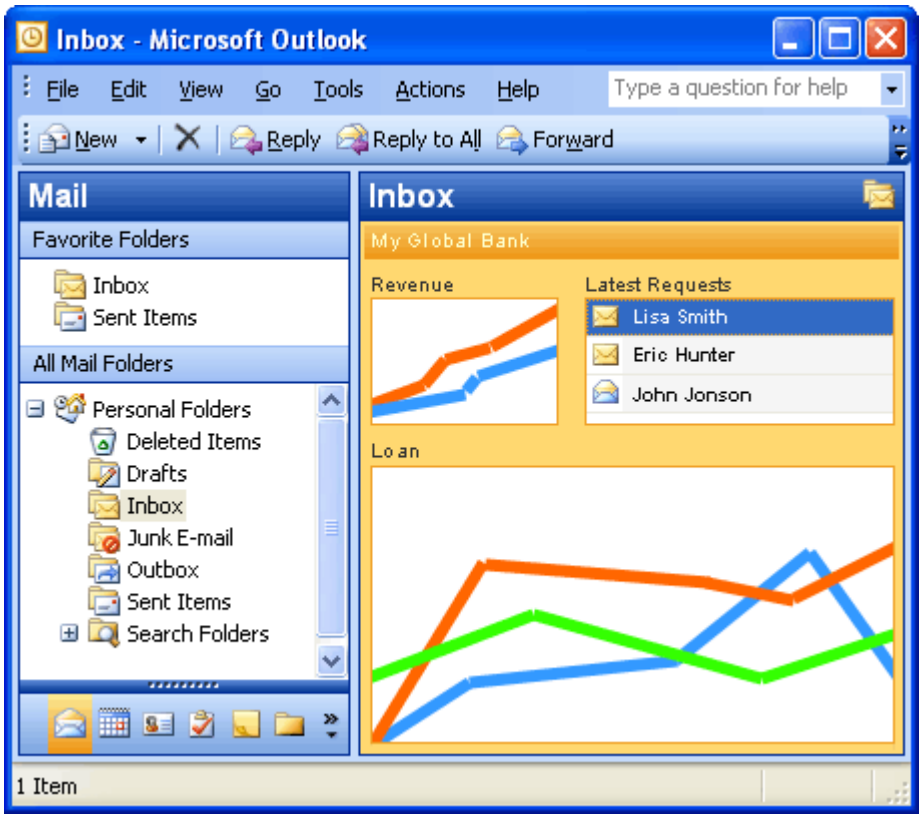
Outlook Regions

Outlook regions are specified in the ExplorerLayout and InspectorLayout properties of the item (= ADXOlFormsCollectionItem). Note that you must also specify the item's ExplorerItemTypes and/or InspectorItemTypes properties; otherwise, the form (an instance of ADXOlForm) will never be shown. Here is the list of Outlook regions:

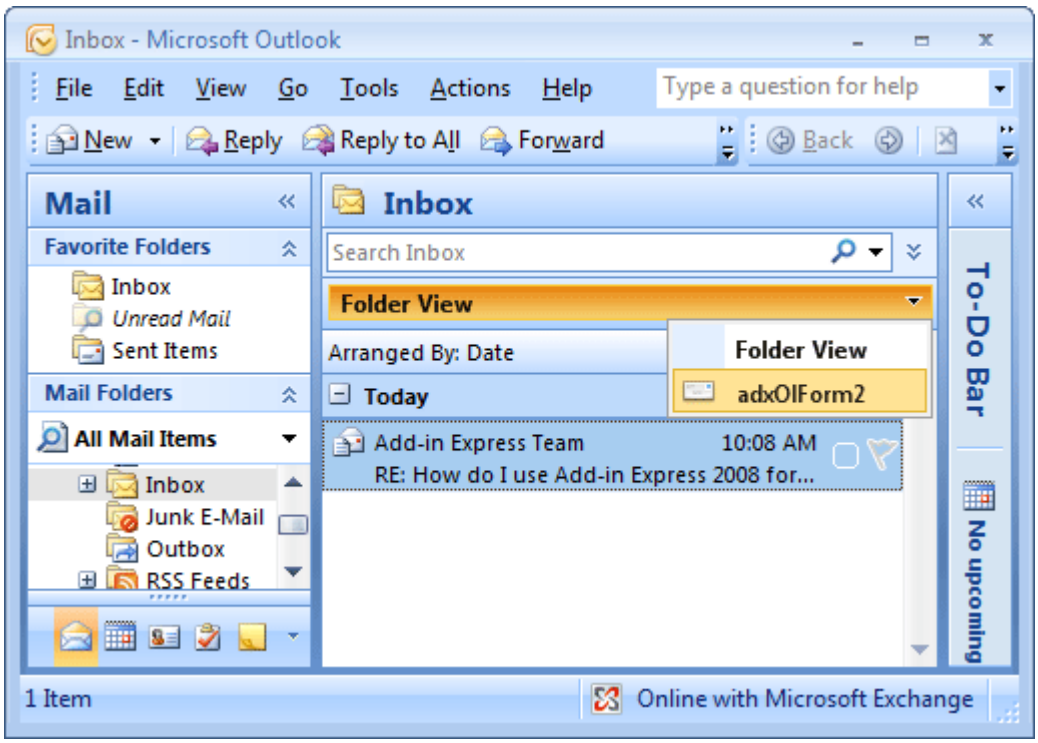
- Four regions around the list of mails, tasks, contacts etc. The region names are LeftSubpane, TopSubpane, RightSubpane, BottomSubpane (see the screenshot below)
- One region below the Navigation Pane – BottomNavigationPane (see the screenshot below)
- One region below the To-Do Bar – BottomTodoBar (see the screenshot below)



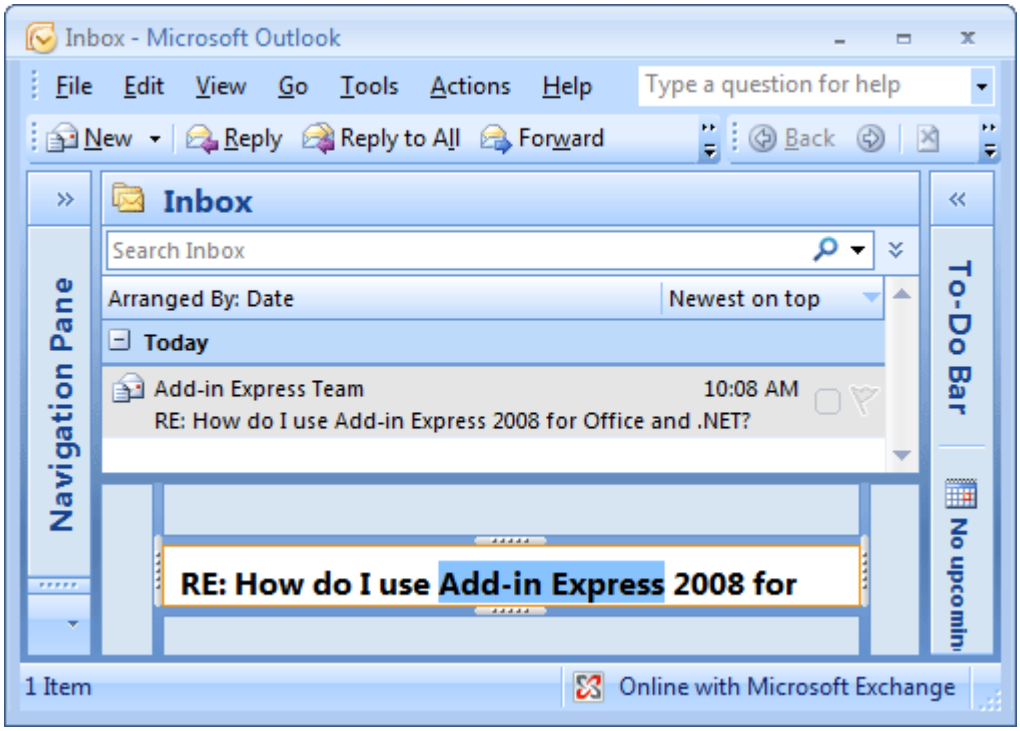
- One region below the Outlook Bar (Outlook 2000 and 2002) – BottomOutlookBar
- The WebViewPane region (see the screenshot below). Note that it uses Outlook properties in order to replace the items grid with your form (see also WebViewPane).



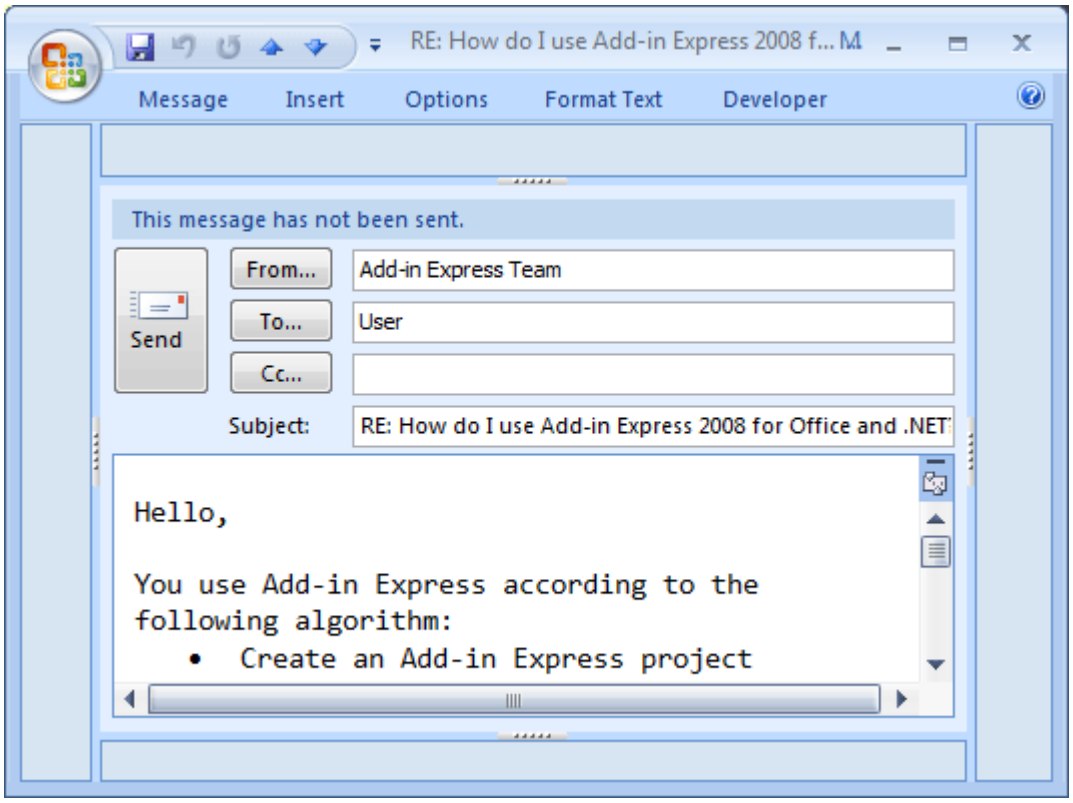
- The FolderView region. Unlike WebViewPane, it allows the user to switch between the original Outlook view and your form.



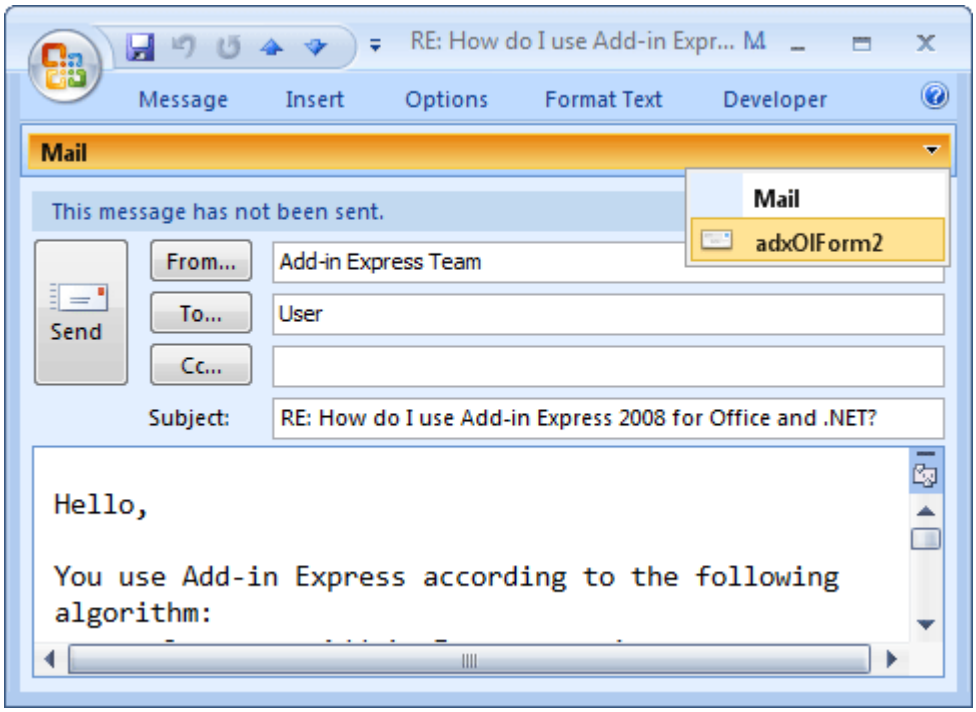
- Four regions around the Reading Pane – LeftReadingPane, TopReadingPane, RightReadingPane, BottomReadingPane (see the screenshot below)



- Four regions around the body of an e-mail, task, contact, etc. The region names are LeftSubpane, TopSubpane, RightSubpane, BottomSubpane (see the screenshot below)



- The InspectorRegion region (see the screenshot below)



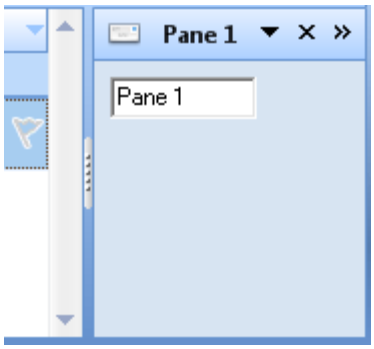
The UI Mechanics

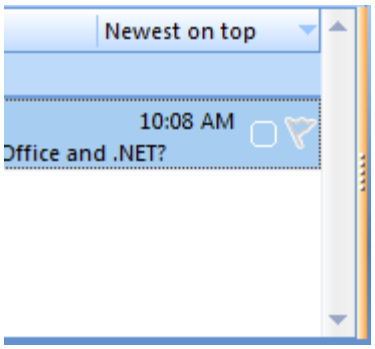
Please read An Absolute Must-Know above before you read the text below.

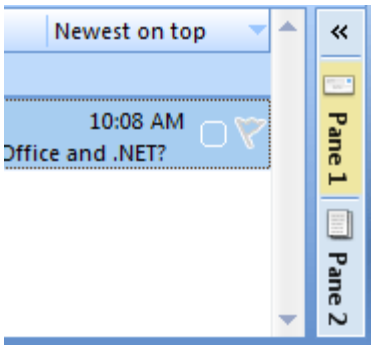
The UI, Related Properties and Events

As mentioned in An Absolute Must-Know, the <Manager> creates instances of the <Form>. An instance of the <Form> (further on it is referenced as form) is considered visible if it is embedded into a region. The form may be actually invisible either due to the region state (see below) or because other forms in the same region hide it; anyway, in this case, <Form>.Visible returns true. To prevent embedding the form into a region, you can set <Form>.Visible to false in the event named ADXBeforeFormShow in Outlook, ADXBeforeTaskPaneShow in Excel, Word, and PowerPoint. When the form is shown in a region, the Activated event occurs and <Form>.Active becomes true. When the user moves the focus onto the form, the <Form> generates the ADXEnter event. When the form loses focus, the ADXLeave event occurs. When the form becomes invisible (actually), it generates the Deactivate event. When the corresponding <Manager> removes the form from its region, <Form>.Visible becomes false and the form generates the ADXAfterFormHide event in Outlook, ADXAfterTaskPaneHide event in Excel, Word, and PowerPoint.

The form may be initially shown in any of the following region states: normal, hidden (collapsed to a 5px wide strip), minimized (reduced to the size of the form caption).







You can change the state of your form at run-time using the <Form>.RegionState property. When showing your Outlook form in some layouts, you need to show the standard form that your form overlays; use the ADXOlForm.ActivateStandardPane() method. Also, you can use the DefaultRegionState property of the <Item>. Note that this property will work for you when you show the form in that region for the very first time and no other forms have been shown in that region before.

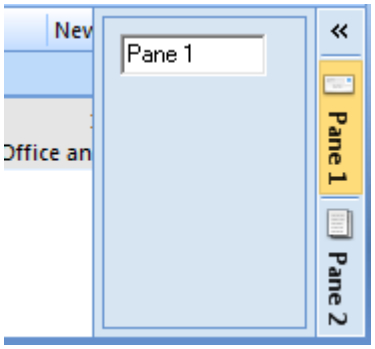
When the region is in the hidden state, the user can click on the splitter and the region will be restored (it will go to the normal state).

When the region is in the normal state, the user can choose any of the options below:

- change the region size by dragging the splitter; this raises size-related events of the form
- hide the form by clicking on the "dotted" mini-button or by double-clicking anywhere on the splitter; this fires the Deactivate event of the <Form>
- close the form by clicking on the Close button in the form header; this fires the ADXCloseButtonClick event of the <Form>. The event is cancellable; if the event isn't cancelled, the Deactivate event occurs, then the pane is being deleted from the region (<Form>.Visible = false) and finally, the <ADXAfterFormHide> event of the <Form> occurs
- show another form by clicking the header and choosing an appropriate item in the popup menu; this fires the Deactivate event on the first form and the Activated event on the second form
- transfer the region to the minimized state by clicking the arrow in the right corner of the form header; this fires the Deactivate event of the form.

When the region is in the minimized state, the user can choose any of the three options below:

- restore the region to the normal state by clicking the arrow at the top of the slim profile of the form region; this raises the Activated event of the form and changes the Active property of the form to true
- expand the form itself by clicking on the form's button; this opens the form so that it overlays a part of the Office application's window near the form region; this also raises the Activated event of the form and sets the Active property of the form to true.
- drag an Outlook item, Excel chart, file, selected text, etc onto the form button; this fires the ADXDragOverMinimized event of the form; the event allows you to check the object being dragged and to decide if the form should be restored.



The Close Button and the Header

The Close button is shown if the CloseButton property of the <Item> is true. The header is always shown when there are two or more forms in the same region. When there is just one form in a region, the header is shown only if the AlwaysShowHeader property of the <Item> is true.

Clicking on the Close button in the form header fires the ADXCloseButtonClick event of the <Form>, the event is cancellable:

```
Private Sub ADXOlForm1_ADXCloseButtonClick(ByVal sender As System.Object, _
    ByVal e As AddinExpress.OL.ADXOlForm.ADXCloseButtonClickEventArgs) _
    Handles MyBase.ADXCloseButtonClick
    e.CloseForm = False
End Sub
```

You can create a Ribbon or command bar button that allows the user to show the form that was previously hidden.

Showing/Hiding Form Instances Programmatically

In Excel and PowerPoint, a single instance of the <Form> is always created for a given <Item> because these applications show documents in a single main window. On the contrary, Word is an application that **normally** shows multiple windows, and in this situation, the Word Task Panes Manager creates one instance of the pane for every document opened in Word.

Outlook is a specific host application. It shows several instances of two window types simultaneously. In addition, the user can navigate through the folder tree and select, create and read several Outlook item types. Accordingly, an ADXOlFormsCollectionItem can generate and show several instances of ADXOlForm at the same time. Find more details on managing custom panes in Outlook in Advanced Outlook Regions.

To access the form, which is currently active in Excel or PowerPoint, you use the TaskPaneInstance property of the <Item>; in Word, the property name is CurrentTaskPaneInstance; in Outlook, it is the GetCurrentForm method. To access all instances of the <Form> in Word, you use the TaskPaneInstances property of ADXWordTaskPanesCollectionItem; in Outlook, you use the FormInstances method of ADXOlFormsCollectionItem. Note that in Excel and PowerPoint an only instance of the <Form> is always created for a given <Item>.

By setting the Enabled property of an <Item> to false, you delete all form instances created for that <Item>. To hide any given form (i.e. to remove it from the region), call its Hide method.

You can check that a form is not available in the UI (say, you cancelled the <BeforInstanceCreate> event or set <Form>.Visible = False in the <BeforeFormShow> event or the user closed it) by checking the Visible property of the form:

```
Dim Pane As ADXWordTaskPane1 = _
    TryCast(Me.AdxWordTaskPanesCollectionItem1.CurrentTaskPaneInstance, _
        ADXWordTaskPane1)
Dim DoesPaneExist As Boolean
If Pane IsNot Nothing Then
    DoesPaneExist = Pane.Visible
Else
    DoesPaneExist = False
End If
```

If the form is not available in the UI, you can show such a form in one step:

- for Outlook, you call the ApplyTo method of the <Item>; the method accepts the parameter, which is either Outlook.Explorer or Outlook.Inspector;
- for Excel, Word, and PowerPoint, you call the ShowTaskPane method of the <Item>

The methods above also transfer the region that shows the form to the normal state.

If the Active property of your form is false, that is if your form is hidden by other forms in the region, then you can call the Activate method of the <Form> to show the form on top of all other forms in that region. If the region was in either minimized or hidden state, calling Activate will also transfer it to the normal state.

Note that your form does not restore its Active state in subsequent sessions of the host application in regions showing several forms. In other words, if several add-ins show several forms in the same region and the current session ends with a given form on top of all other forms in that region, the subsequent start of the host application may show some other form as active. This is because events are given to add-ins in an unpredictable order. When dealing with several forms of a given add-in, they are created in the order determined by their locations in the <Items> collection of the <Manager>.

In Outlook, due to context-sensitivity features provided by the <Item>, an instance of your form will be created whenever the current context matches that specified by the corresponding <Item>.

Resizing the Forms

There are two values of the Splitter property of the <Item>. The default one is Standard. This value shows the splitter allowing the user to change the form size as required. The form size is stored in the registry so that the size is restored whenever the user starts the host application.

You can only resize your form programmatically, if you set the Splitter property to None. This prevents the user form resizing the form. Changing the Splitter property at run time does not affect a form currently loaded into its region (that is, having Visible = true). Instead, it will be applied to any newly shown form.

If the form is shown in a given region for the first time and no forms were ever shown in this region, the form will be shown using the appropriate dimensions that you set at design-time. On subsequent host application sessions, the form will be shown using the dimensions set by the user.

Tuning the Settings at Run-Time

To add/remove an <Item> to/from the collection and to customize the properties of an <Item> at add-in start-up, you use the <Initialize> event of the <Manager>; the event's name is OnInitialize for Outlook and ADXInitalize for Excel, Word and PowerPoint.

Changing the Enable, Cached (Outlook only), <FormClassName> properties at run-time deletes all form instances created by the <Item>.

Changing the InspectorItemTypes, ExplorerItemTypes, ExplorerMessageClasses, ExplorerMessageClass, InspectorMessageClasses, InspectorMessageClass, FolderNames, FolderName properties of the ADXOlFormsCollectionItem deletes all non-visible form instances.

Changing the <Position> property of the <Item> changes the position for all visible form instances.

Changing the Splitter and Tag properties of the <Item> doesn't do anything for the currently visible form instances. You will see the changed splitter when the <Manager> shows a new instance of the <Form>.

Excel Task Panes

Please see The UI Mechanics above for the detailed description of how Add-in Express panes work. Below you see the list containing some generic terms mentioned in An Absolute Must-Know and their Excel-specific equivalents:

- <Manager> - AddinExpress.XL.ADXExcelTaskPanesManager, the Excel Task Panes Manager
- <Item> - AddinExpress.XL.ADXExcelTaskPanesCollectionItem
- <Form> - AddinExpress.XL.ADXExcelTaskPane

Application-specific features

ADXExcelTaskPane provides useful events unavailable in the Excel object model: ADXBeforeCellEdit and ADXAfterCellEdit.

Keyboard and Focus

ADXExcelTaskPane provides the ADXKeyFilter event. It deals with the feature of Excel that captures the focus if a key combination handled by Excel is pressed. By default, Add-in Express panes do not pass key combinations to Excel. In this way, you can be sure that the focus will not leave the pane unexpectedly.

Just to understand that Excel feature, imagine that you need to let the user press Ctrl+S and get the workbook saved while your pane is focused. In such a scenario, you have two ways:

- You process the key combination in the code of the pane and use the Excel object model to save the workbook.
- Or you send this key combination to Excel using the ADXKeyFilter event.

Besides the obvious difference between the two ways above, the former leaves the focus on your pane while the latter effectively moves it to Excel because of the focus-capturing feature just mentioned.

The algorithm of key processing is as follows. Whenever a single key is pressed, it is sent to the pane. When a key combination is pressed, ADXExcelTaskPane determines if the combination is a shortcut on the pane. If it is, the keystroke is sent to the pane. If it isn't, ADXKeyFilter is fired and the key combination is passed to the event handler. Then the event handler specifies whether to send the key press to Excel or to the pane. The

latter is the default behavior. Note that sending the key combination to Excel will result in moving the focus off the pane. The above-said implies that the ADXKeyFilter event never fires for shortcuts on the pane's controls.

ADXKeyFilter is also never fired for hot keys (Alt + an alphanumeric symbol). If ADXExcelTaskPane determines that the pane cannot process the hot key, it sends the hot key to Excel, which activates its main menu. After the user has navigated through the menu by pressing arrow buttons, Esc, and other hot keys, opened and closed Excel dialogs, ADXExcelTaskPane will get focus again.

Wait a Little and Focus Again

The pane provides a simple infrastructure that allows implementing the Wait a Little schema - see the ADXPostMessage method and the ADXPostMessageReceived event.

Currently we know at least one situation when this trick is required. Imagine that you show a pane and you need to set the focus on a control on the pane. It isn't a problem to do this in, say, the Activated event. Nevertheless, it is useless because Excel, continuing its initialization, moves the focus off the pane. With the above-said method and event, you can make your pane look like it never loses focus: in the Activated event handler, you call the ADXPostMessage method specifying a unique message ID and, in the ADXPostMessageReceived event, you filter incoming messages. When you get the appropriate message, you set the focus on the control. Beware, there will be a huge lot of inappropriate messages in the ADXPostMessageReceived event.

Advanced Outlook Regions

Please see The UI Mechanics above for the detailed description of how Add-in Express panes work. Below you see the list containing some generic terms mentioned in An Absolute Must-Know and their Outlook-specific equivalents:

- <Manager> - AddinExpress.OL.ADXOlFormsManager, the Outlook Forms Manager
- <Item> - AddinExpress.OL.ADXOlFormsCollectionItem
- <Form> - AddinExpress.OL.ADXOlForm

Context-Sensitivity of Your Outlook Form

Whenever the Outlook Forms Manager detects a context change in Outlook, it searches the ADXOlFormsCollection collection for enabled items that match the current context and, if any match is found, it shows or creates the corresponding instances.

ADXOlFormsCollectionItem provides a number of properties that allow specifying the context settings for your form. Say, you can specify **item types** for which your form will be shown. Note that in case of explorer, the item types that you specify are compared with the default item type of the current folder. In addition, you can specify

the names of the folders for which your form will be shown in the FolderName and FolderNames properties; these properties also work for Inspector windows – in this case, the parent folder of the Outlook item is checked. A special value in FolderName is an asterisk ('*'), which means "all folders". See also COM Add-ins for Outlook – Template Characters in FolderName. You can also specify **message class(es)** for which your form will be shown. Note that all context-sensitivity properties of an ADXOlFormsCollectionItem are processed using the **OR** Boolean operation.

In advanced scenarios, you can also use the ADXOlFormsCollectionItem.ADXBeforeFormInstanceCreate and ADXOlForm.ADXBeforeFormShow events in order to prevent your form from being shown (see Showing/Hiding Form Instances Programmatically). In addition, you can use events provided by ADXOlForm in order to check the current context. Say, you can use the ADXBeforeFolderSwitch or ADXSelectionChange events of ADXOlForm.

Caching Forms

By default, whenever Add-in Express needs to show a form, it creates a new instance of that form. You can change this behavior by choosing an appropriate value of the ADXOlFormsCollectionItem.Cached property. The values of this property are:

- NewInstanceForEachFolder – it shows the same form instance whenever the user navigates to the same Outlook folder.
- OneInstanceForAllFolders – it shows the same form instance for all Outlook folders.
- None – no form caching is used.

Caching works within the same Explorer window: when the user opens another Explorer window, Add-in Express creates another set of cached forms. Forms shown in Inspector windows cannot be cached.

Is It Inspector or Explorer?

Check the InspectorObj and ExplorerObj properties of ADXOlForm. These properties return COM objects that will be released when your form is removed from its region. This may occur several times during the lifetime of a given form instance because Add-in Express may remove your form from a given region and then embed the form to the same region in order to comply with Outlook windowing.

WebViewPane

When this value (see Outlook Regions) is chosen in the ExplorerLayout property of ADXOlFormsCollectionItem, Add-in Express uses the WebViewUrl and WebViewOn properties of Outlook.MAPIFolder (also Outlook.Folder in Outlook 2007-2010) in order to show your form as a home page for a given folder(s).

Unfortunately, due to a bug in Outlook 2002, Add-in Express has to scan all Outlook folders in order to set and restore the WebViewUrl and WebViewOn properties. The first consequence is a delay at startup if the current profile contains thousands of folders. A simple way to prevent the delay is to disable the corresponding item(s) of the Items collection of the Outlook Forms Manager at design-time and enable it in the AddinStartupComplete event of the add-in module. Because *PublicFolders* usually contains many folders, Add-in Express doesn't allow using WebViewPane for *PublicFolders* and all folders below it. *Outbox* and *Sync Issues* and all folders below them aren't supported as well when using WebViewPane.

Because of the need to scan Outlook folders, WebViewPane produces another delay when the user works in the Cached Exchange Mode (see the properties of the Exchange account in Outlook) and the Internet connection is slow or broken. To bypass this problem Add-in Express allows reading EntryIDs of those folders from the registry. Naturally, you are supposed to write appropriate values to the registry at add-in start-up. Here is the code to be used in the add-in module:

```
internal void SaveDefaultFoldersEntryIDToRegistry(string PublicFoldersEntryID,
    string PublicFoldersAllPublicFoldersEntryID,
    string FolderSyncIssuesEntryID)
{
    RegistryKey ModuleKey = null;
    RegistryKey ADXXOLKey = null;
    RegistryKey WebViewPaneSpecialFoldersKey = null;
    try
    {
        ModuleKey = Registry.CurrentUser.OpenSubKey(this.RegistryKey, true);
        if (ModuleKey != null)
        {
            ADXXOLKey = ModuleKey.CreateSubKey("ADXXOL");
            if (ADXXOLKey != null)
            {
                WebViewPaneSpecialFoldersKey =
                    ADXXOLKey.CreateSubKey
                    ("FoldersForExcludingFromUseWebViewPaneLayout");
                if (WebViewPaneSpecialFoldersKey != null)
                {
                    if (PublicFoldersEntryID.Length >= 0)
                    {
                        WebViewPaneSpecialFoldersKey.
                            SetValue("PublicFolders",
                            PublicFoldersEntryID);
                    }
                    if (PublicFoldersAllPublicFoldersEntryID.Length >= 0)
                    {
                        WebViewPaneSpecialFoldersKey.
                            SetValue("PublicFoldersAllPublicFolders",
                            PublicFoldersAllPublicFoldersEntryID);
                    }
```

```
                    if (FolderSyncIssuesEntryID.Length >= 0)
                    {
                        WebViewPaneSpecialFoldersKey.
                            SetValue("FolderSyncIssues",
                            FolderSyncIssuesEntryID);
                    }
                }
            }
        }
    }
    finally
    {
        if (ModuleKey != null)
        {
            ModuleKey.Close();
        }
        if (WebViewPaneSpecialFoldersKey != null)
        {
            WebViewPaneSpecialFoldersKey.Close();
        }
        if (ADXXOLKey != null)
        {
            ADXXOLKey.Close();
        }
    }
}
```

Toolbar Controls for Microsoft Office

The Add-in Express Extensions for Microsoft Office Toolbars (or the Toolbar Controls) is a plug-in for Add-in Express designed to overstep the limits of existing CommandBar controls. With the Toolbar Controls, you can use any .NET controls, not only Office controls, on your command bars. Now you can add tree-views, grids, diagrams, edit boxes, reports, etc. to your command bars.

To make the text below easy to read, let's define three terms, namely:

- Command bar controls are controls such as command bar buttons and command bar combo boxes provided by the Office object model. These controls are Office controls and they are supported by Add-in Express.
- Non-Office controls are any controls, both .NET built-in and third party controls, such as tree-views, grids, user controls, etc. Usually, you use these controls on your Windows application forms.
- Advanced command bar control is an instance of ADXCommandBarAdvancedControl or the ADXCommandBarAdvancedControl class itself (depending on the context).

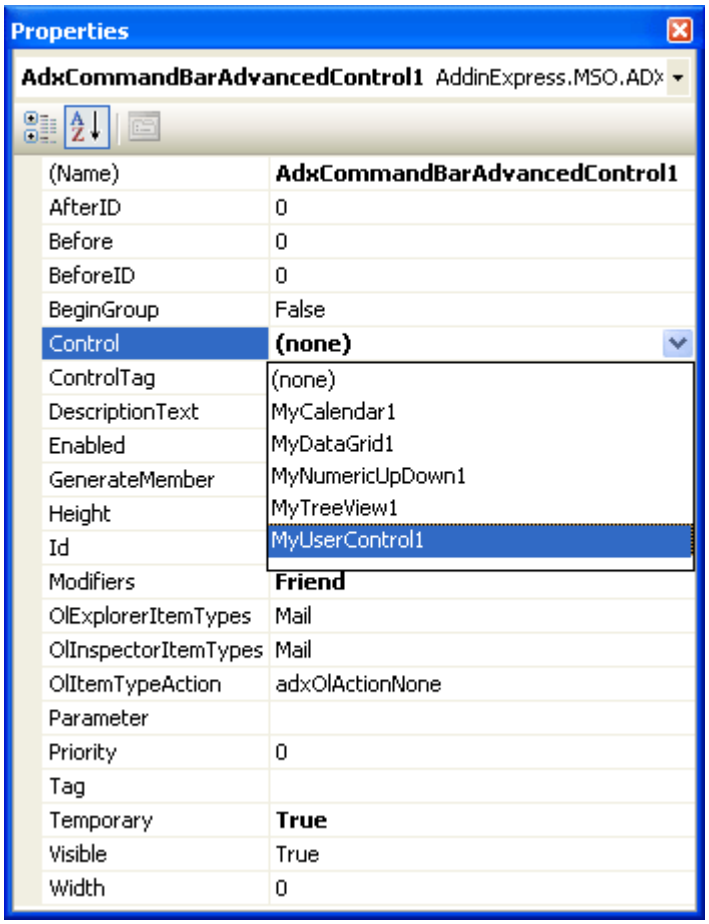
What is ADXCommandBarAdvancedControl

If you have developed at least one add-in based on Add-in Express, you probably ran into ADXCommandBarAdvancedControl when adding command bar controls to your command bars. Yes, it is that strange item of the Add button on the ADXCommandBarControl collection editor.

This plug-in gives you a chance to use any non-Office controls such as tree-views, grids, labels, edit and combo boxes, diagrams on any Office command bars. Now you can add ADXCommandBarAdvancedControl, an advanced command bar control, to your command bar and bind it to any non-Office control you placed on the add-in module. As a result, you will have your grid, tree-view or image placed on your command bar.

Hosting any .NET Controls

In addition to properties common for Office command bar controls, ADXCommandBarAdvancedControl has one more property. It is the Control property, the most important one. With this property, you can select a non-Office control to place it on your command bar. Have a look at the picture below. The add-in module contains five controls – MyCalendar, MyDataGrid, MyNumericUpDown, MyTreeView and MyUserControl. The Control property asks you to select one of these controls. If you select MyUserControl, your add-in adds MyUserControl to your command bar. With the Control property, ADXCommandBarAdvancedControl becomes a host for your non-Office controls.

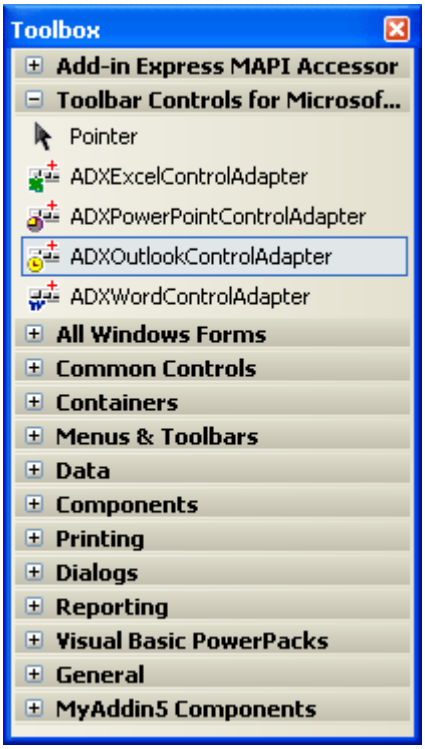


On .NET, ADXCommandBarAdvancedControl supports all controls based on System.Windows.Forms.Control. Therefore, on your command bars, you can use both built-in controls and third-party controls based on System.Windows.Forms.Control. Just add them to the add-in module, add an advanced command bar control to your command bar, and select your non-Office control in the Control property of ADXCommandBarAdvancedControl.

Control Adapters

You may ask us what the Toolbar Controls described above does and what it is for, if ADXCommandBarAdvancedControl is already included in Add-in Express. In general, ADXCommandBarAdvancedControl is still abstract in Add-in Express but it is implemented by the Toolbar Controls if it is plugged in Add-in Express. So, the answer is: the Toolbar Controls for Microsoft Office implements ADXCommandBarAdvancedControl for each Office application.

The Toolbar Controls adds a new tab, "Toolbar Controls for Microsoft Office", to the Toolbox and places several components on the tab (see the screenshot below). The Toolbar Controls supports each Office application by special components called control adapters. Only control adapters know how to add your controls to applications specific command bars. So, the control adapters are the Toolbar Controls itself.



In Express editions of Visual Studio, you need to add the control adapters manually.

The add-in module can contain control adapters only. For example, you should add an ADXExcelControlAdapter to the add-in module if you want to use non-Office controls in your Excel add-in. To use non-Office controls on several Office applications you should add several control adapters. For example, if you need to use your controls in your add-in that supports Outlook, Excel, and Word, you should add three control adapters: ADXExcelControlAdapter, ADXWordControlAdapter, and ADXOutlookControlAdapter to the add-in module.

ADXCommandBarAdvancedControl

As described above, the Toolbar Controls implements the ADXCommandBarAdvancedControl class that is still abstract in Add-in Express without the Toolbar Controls installed. In addition to properties common for all command bar controls, ADXCommandBarAdvancedControl provides two special properties related to the Toolbar Controls.

The Control Property

The Control property binds its ADXCommandBarAdvancedControl to a non-Office control; it can be used at design-time as well as at run-time. To place your non-Office control on your command bar you just select your control in the Control property at design-time, or set the Control property to an instance of your control at run-time:

```
Private Sub AddinModule_AddinInitialize( _
   ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles MyBase.AddinInitialize
   BossCheckbox = New System.Windows.Forms.CheckBox
   Me.AdxCommandBarAdvancedControl1.Control = BossCheckbox
End Sub
```

The ActiveInstance Property

The ActiveInstance property is read-only; it returns the instance of the control that was created for the current context. For example, you can initialize your control for the active Inspector window by handling the InspectorActivate event:

```
Private Sub adxOutlookEvents_InspectorActivate( _
   ByVal sender As System.Object, ByVal inspector As System.Object,
   ByVal folderName As System.String) _
      Handles adxOutlookEvents.InspectorActivate

   Dim ChkBox As System.Windows.Forms.CheckBox = _
      Me.AdxCommandBarAdvancedControl1.ActiveInstance
   If ChkBox IsNot Nothing Then ChkBox.Enabled = False
End Sub
```

Please note that the ActiveInstance property is not valid in most cases when you may want to use it. However, you can always use any window activate events such as the InspectorActivate event of Outlook and WindowActivate event of Word. The table bellow shows you the order of event processing by the example of the Outlook Inspector window opened by the user.

1. Outlook fires the built-in NewInspector event. Add-in Express traps it and fires the NewInspector event of ADXOutlookEvents.		ActiveInstance returns NULL.
	2. ADXOutlookEvents runs your NewInspector event handlers.	ActiveInstance returns NULL.
	3. The Toolbar Controls creates an instance of your control.	ActiveInstance returns NULL.
	4. Outlook fires the built-in InspectorActivate event. Add-in Express handles it and fires the InspectorActivate event of ADXOutlookEvents.	ActiveInstance returns NULL.
	5. The Toolbar Controls creates an instance of your control for the opened Inspector. ADXOutlookEvents runs your InspectorActivate event handlers.	ActiveInstance returns the instance of your control that was cloned from your original control.

Application-specific Control Adapters

All Office applications have different window architectures. All Office windows themselves are different. All our control adapters have a unified programming interface but different internal architectures that take into account the windows architecture of the corresponding applications. All features of all control adapters are described below.

Outlook

Outlook has two main windows – Explorer and Inspector windows. The user can open several Explorer and Inspector windows. Our Outlook control adapter supports non-Office controls on both Explorer and Inspector windows, and creates an instance of your control whenever the user opens a new window.

Please note, if Word is used as an e-mail editor, Outlook uses MS Word as an Inspector window. In this case, Word is running in a separate process. In this scenario, because of obvious and unsolvable problems the Outlook control adapter hides all instances of your control on all inactive Word Inspector windows, but shows them once the Inspector is activated.

Excel

In spite of the fact that Excel allows placing its windows on the Task Bar, all its command bars work like in MDI applications. Therefore, your controls are created only once, at Excel start-up. However, you can still use the WorkbookActivate, WindowActivate, and SheetActivate events to initialize your non-Office controls according to the context.

Word

Word creates its command bars for all document windows, so your non-Office controls are instanced whenever the user opens a new window or a document. We recommend using the WindowActivate event to initialize your control for the current window.

PowerPoint

Notwithstanding the fact that PowerPoint makes possible placing its windows on the Task Bar, PowerPoint is an MDI application. Therefore, your controls are created only once, at PowerPoint startup. However, you can still use the WindowActivate event to initialize your non-Office controls according to the context.

Samples

See Your First .NET Control on an Office Toolbar. See also the HOWTOs section on our web site.

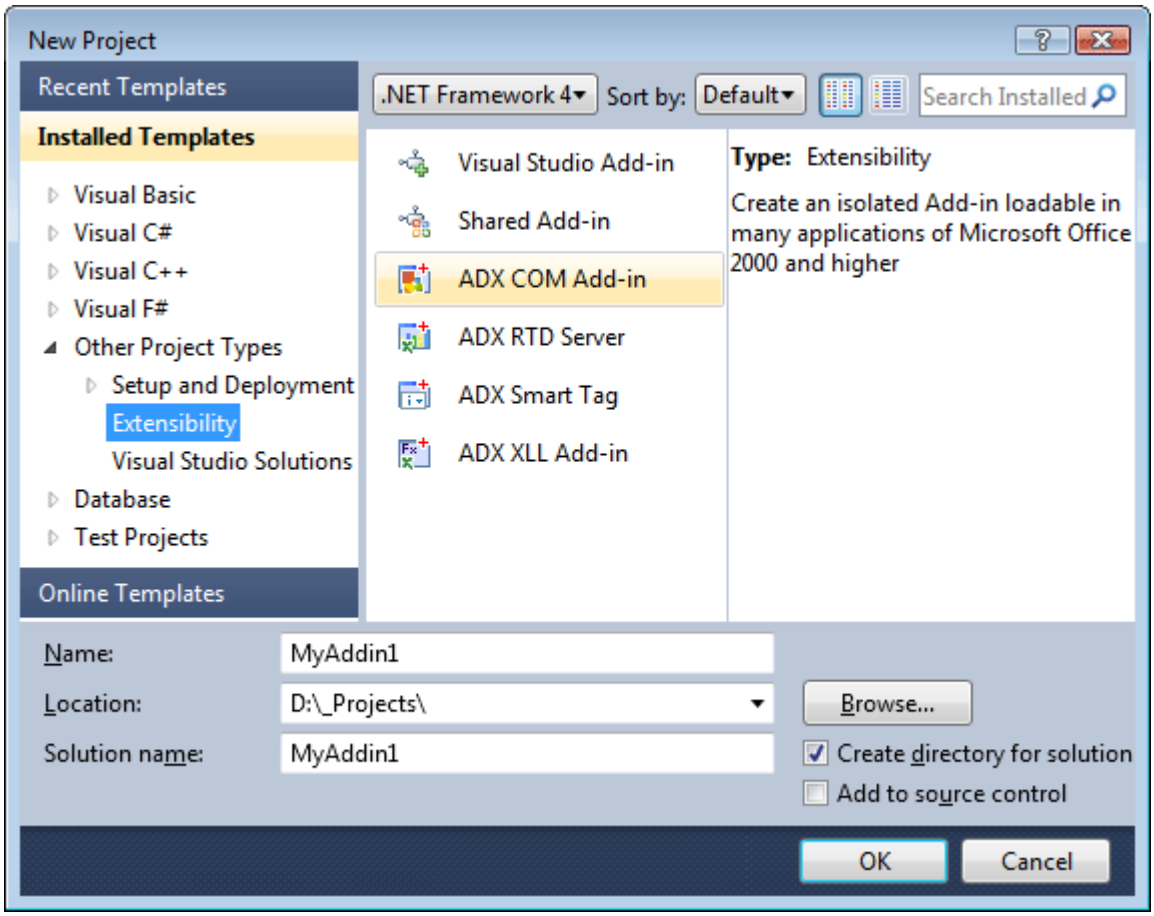
Sample Projects

Your First Microsoft Office COM Add-in

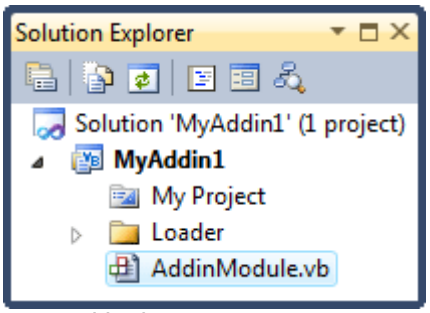
This VB.NET sample project implements a COM add-in for Excel, Word and PowerPoint versions 2000-2010.

Step #1 - Creating a COM Add-in Project

Start Visual Studio via "Run as Administrator". Choose *Add-in Express COM Add-in* in the New Project dialog.



Click OK to start the COM add-in project wizard. In the wizard, you choose the programming language of your add-in, as well as interop assemblies to use and Office applications to support in your add-in, see Choosing Interop Assemblies.

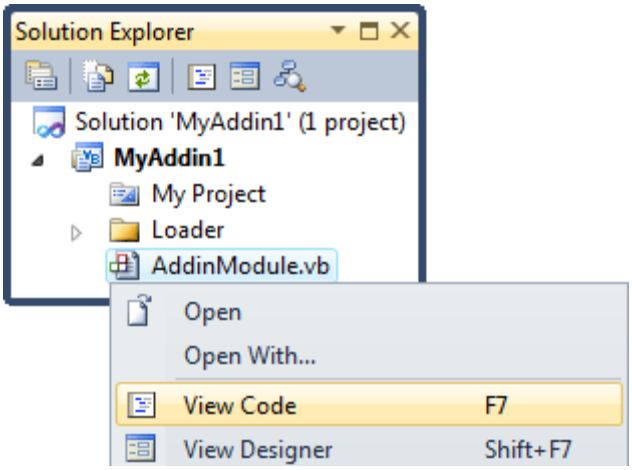


The project wizard creates and opens a new solution in the IDE. The solution contains an only project, the add-in project.

The add-in project contains the AddinModule.vb (or AddinModule1.cs) file discussed in the next step.

Step #2 - Add-in Module

AddinModule.vb (or AddinModule1.cs) is the core part of the add-in project. It is a container for components essential for the functionality of your add-in. You specify the add-in properties in the module's properties, add the components to the module's designer, and write the functional code of your add-in in this module. To review its source code, in Solution Explorer, right-click the AddinModule1.vb (or AddinModule1.cs) file and choose *View Code* in the popup menu.



The code for AddinModule1.vb is as follows:

```
Imports System.Runtime.InteropServices
Imports System.ComponentModel

'Add-in Express Add-in Module
<GuidAttribute("888782EF-544A-4EDD-8977-D24C4EE6F04D"),
ProgIdAttribute("MyAddin1.AddinModule")> _
Public Class AddinModule
    Inherits AddinExpress.MSO.ADXAddinModule

#Region " Component Designer generated code. "
    'Required by designer
    Private components As System.ComponentModel.IContainer

    'Required by designer - do not modify
    'the following method
    Private Sub InitializeComponent()
        Me.components = New System.ComponentModel.Container()
        '
        'AddinModule
        '
        Me.AddinName = "MyAddin1"

        Me.SupportedApps = CType(( _
                  AddinExpress.MSO.ADXOfficeHostApp.ohaExcel Or _
                  AddinExpress.MSO.ADXOfficeHostApp.ohaWord Or _
                  AddinExpress.MSO.ADXOfficeHostApp.ohaPowerPoint  _
                  ), AddinExpress.MSO.ADXOfficeHostApp)
    End Sub

#End Region

#Region " Add-in Express automatic code "
```

```
    'Required by Add-in Express - do not modify
    'the methods within this region

    Public Overrides Function GetContainer() As System.ComponentModel.IContainer
        If components Is Nothing Then
            components = New System.ComponentModel.Container
        End If
        GetContainer = components
    End Function

    <ComRegisterFunctionAttribute()> _
    Public Shared Sub AddinRegister(ByVal t As Type)
        AddinExpress.MSO.ADXAddinModule.ADXRegister(t)
    End Sub

    <ComUnregisterFunctionAttribute()> _
    Public Shared Sub AddinUnregister(ByVal t As Type)
        AddinExpress.MSO.ADXAddinModule.ADXUnregister(t)
    End Sub

    Public Overrides Sub UninstallControls()
        MyBase.UninstallControls()
    End Sub

#End Region

    Public Sub New()
        MyBase.New()

        'This call is required by the Component Designer
        InitializeComponent()

        'Please add any initialization code to the AddinInitialize event handler

    End Sub

    Public ReadOnly Property ExcelApp() As Excel._Application
        Get
            Return CType(HostApplication, Excel._Application)
        End Get
    End Property

    Public ReadOnly Property PowerPointApp() As PowerPoint._Application
        Get
            Return CType(HostApplication, PowerPoint._Application)
        End Get
```

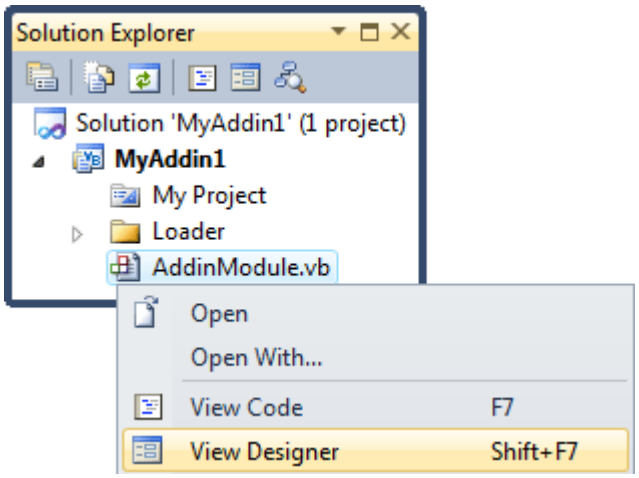
```
    End Property

    Public ReadOnly Property WordApp() As Word._Application
        Get
            Return CType(HostApplication, Word._Application)
        End Get
    End Property

End Class
```

Pay attention to the ExcelApp, PowerPointApp and WordApp properties of the module generated by the wizard. You use them in your code to access the object model(s) of the host application(s) of your add-in.

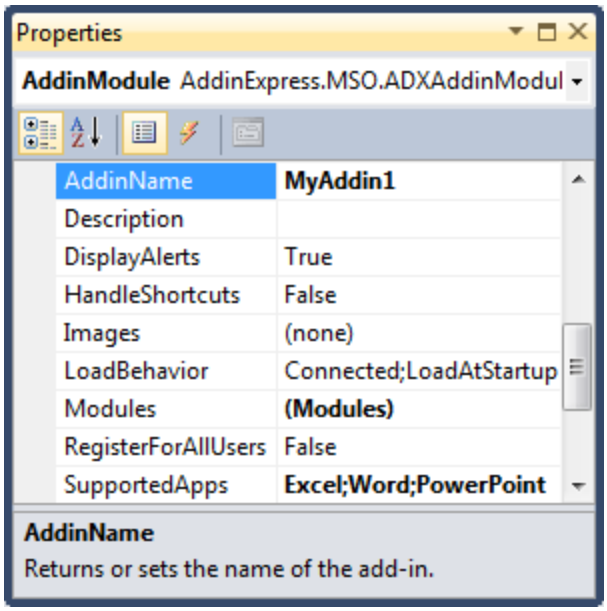
Step #3 – Add-in Module Designer

The designer of the add-in module allows setting add-in properties and adding components to the module.

In Solution Explorer, right-click the AddinModule.vb (or AddinModule.cs) file and choose *View Designer* in the popup menu.



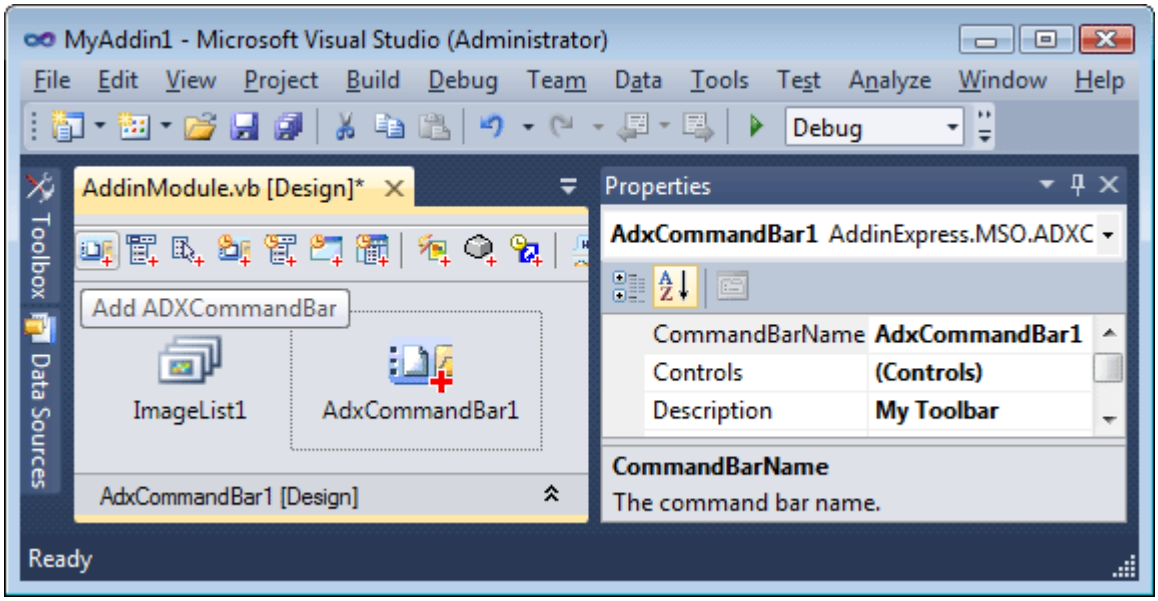
In the *Properties* window, you set the name and description of your add-in.



To add an Add-in Express component to the module, choose an appropriate command in the Commands Toolbar, or you can right-click the designer surface and choose the same command in the context menu. See also Add-in Express Basics and Commands of the Add-in Module.

Step #4 - Adding a New Toolbar

To add a toolbar to your add-in, you use the *Add ADXCommandBar* button. It adds an ADXCommandBar component to the add-in module.



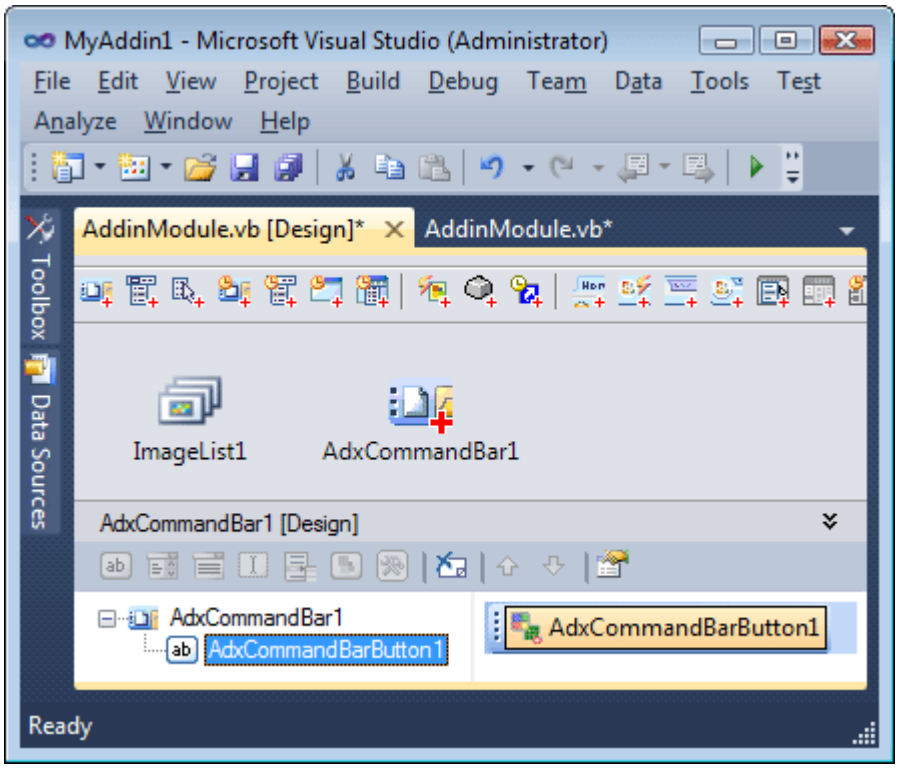
Select the command bar component and, in the *Properties* window, specify the toolbar name in the CommandBarName property.

If the toolbar name is not the same as the name of any built-in command bar of the host application, then the component will create a new toolbar at run-time. That is, if you set CommandBarName = "Standard", and add, say, an ADXCommandBarButton to the Controls collection of the ADXCommandBar component, this will create a button on the built-in *Standard* toolbar, while specifying CommandBarName = "Standard2" will create a new toolbar, *Standard2*, with a button on it. If the *Standard2* toolbar is already present in the host application, the button will be added to that toolbar.

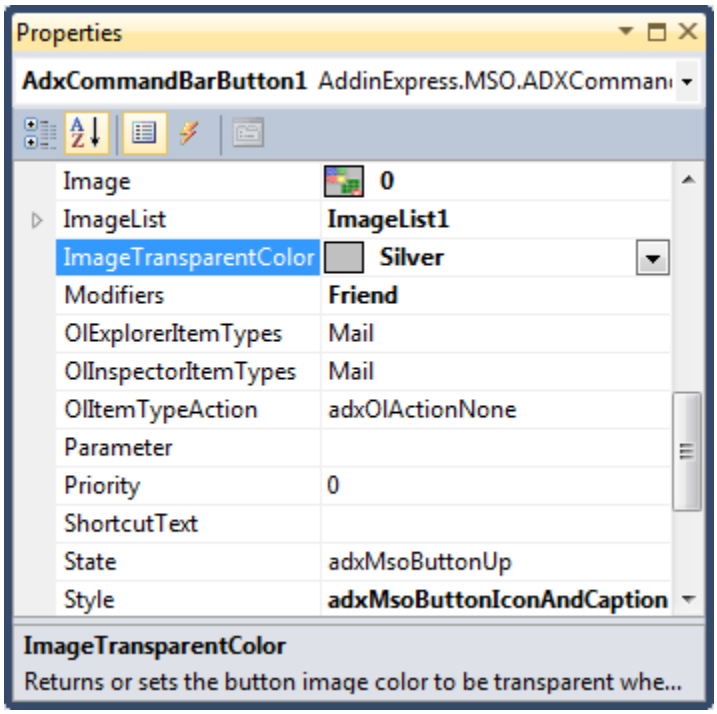
See also Add-in Express Basics and Command Bar UI.

Step #5 - Adding a New Toolbar Button

Select the command bar component on the designer of the add-in module and open the in-place designer area. In this area, you'll see the visual designer of the ADXCommandBar component. Use its toolbar to add or remove command bar controls. Just click the appropriate button and see the result.



In the screenshot above, a toolbar button is already added. Now, select the button and open the *Properties* window where you specify the button's Caption property, change the Style property if you need to show an icon on the button (default value = adxMsoButtonCaption), and add an event handler to the Click event.



In the screenshot on the right, we demonstrate the button properties that make the icon visible and transparent: Style, Image, ImageList and ImageTransparentColor.

Step #6 – Accessing Host Application Objects

The add-in module provides the HostApplication property that returns the Application object (of the Object type) of the host application the add-in is currently running in. For your convenience, the project wizard adds

host-related properties to the module such as ExcelApp, WordApp, etc. You use these properties as entry points to the host applications object models. See how these properties are used in the code below:

```
Private Sub DefaultAction(ByVal sender As System.Object) _
    Handles AdxCommandBarButton1.Click
    MsgBox(GetInfoString())
End Sub

Friend Function GetInfoString() As String
    Dim ActiveWindow As Object = Nothing
    Try
        ActiveWindow = Me.HostApplication.ActiveWindow() 'late binding
    Catch
    End Try
    Dim Result As String = "No document window found!"
    If Not ActiveWindow Is Nothing Then
        Select Case Me.HostType
            Case ADXOfficeHostApp.ohaExcel
                Dim ActiveCell As Excel.Range = _
                    CType(ActiveWindow, Excel.Window).ActiveCell
                If ActiveCell IsNot Nothing Then
                    'relative address
                    Dim Address As String = ActiveCell.AddressLocal(False, False)
                    Marshal.ReleaseComObject(ActiveCell)
                    Result = "The current cell is " + Address
                End If
            Case ADXOfficeHostApp.ohaWord
                Dim Selection As Word.Selection = _
                    CType(ActiveWindow, Word.Window).Selection
                Dim Range As Word.Range = Selection.Range
                Dim Words As Word.Words = Range.Words
                Dim WordCountString = Words.Count.ToString()
                Marshal.ReleaseComObject(Selection)
                Marshal.ReleaseComObject(Range)
                Marshal.ReleaseComObject(Words)
                Result = "There are " + WordCountString _
                    + " words currently selected"
            Case ADXOfficeHostApp.ohaPowerPoint
                Dim Selection As PowerPoint.Selection = _
                    CType(ActiveWindow, PowerPoint.DocumentWindow).Selection
                Dim SlideRange As PowerPoint.SlideRange = Selection.SlideRange
                Dim SlideCountString = SlideRange.Count.ToString()
                Marshal.ReleaseComObject(Selection)
                Marshal.ReleaseComObject(SlideRange)
                Result = "There are " + SlideCountString _
                    + " slides currently selected"
```

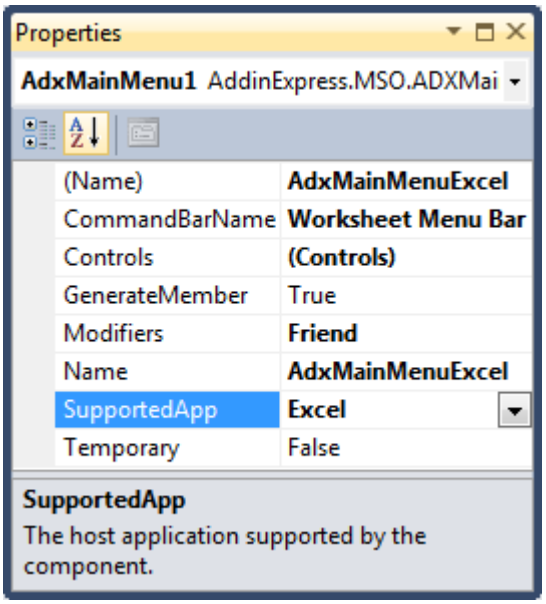
```
            Case Else
                Result = AddinName + " doesn't support " + HostName
        End Select
        Marshal.ReleaseComObject(ActiveWindow)
    End If
    Return Result
End Function
```

Two things in the code above deserve your attention. First, the `GetInfoString` method will be called from a number of events. If this code is run in Word and there is no open document, `Me.HostApplication.ActiveWindow()` will fire an exception. That is why this code line is wrapped in a try/catch block. Second, you have to release every COM object created in your code. See **Error! Reference source not found.** for more details.

Step #7 - Customizing Main Menus

Add-in Express provides a component to customize the main menu of any Office application. Note that several Office applications from Office 2000-2003 have several main menus. Say, in these Excel versions, you find *Worksheet Menu Bar* and *Chart Menu Bar*. Naturally, in Excel 2007 and 2010 these menus are replaced with the Ribbon UI. Nevertheless, they are still accessible programmatically and you may want to use this fact in your code. As for customizing main menus in Outlook, see Your First Microsoft Outlook COM Add-in.

In this sample, we are going to customize the *File* menu in Excel and Word, version 2000-2003. You start with adding two main menu components and specifying correct host applications in their SupportedApp properties. Then, in the CommandBarName property, you specify the main menu.

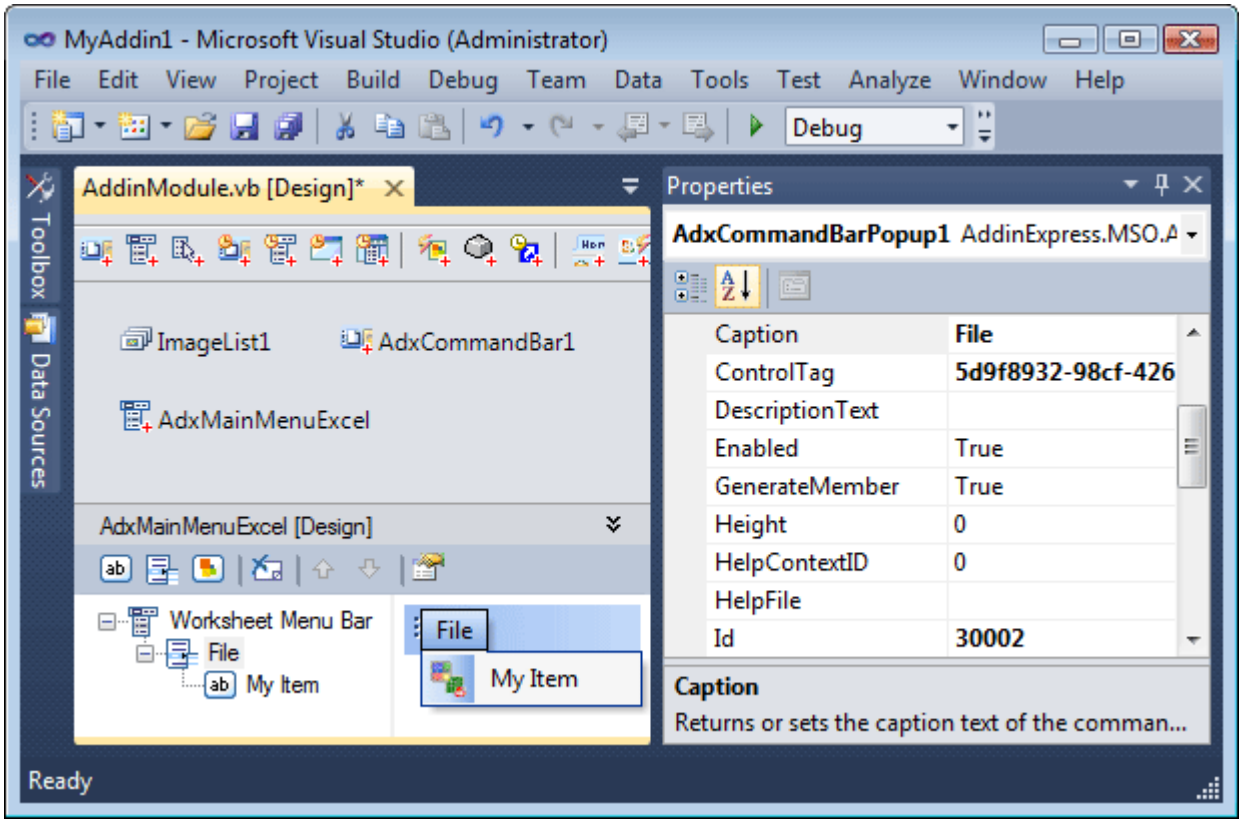


The screenshot on the right shows how you set up the main menu component in order to customize the *Worksheet Menu Bar* main menu in Excel 2000-2003.

Now you can open the in-place designer for the main menu component and populate it with controls. First off, you add a popup control and set its Id property to 30002. Specifying anything but 1 in this property means that controls added to that ADXCommandBarPopup component, will be created on the built-in popup control having ID=30002, which is the ID of the *File* menu item in Office applications. To find this and similar IDs, use our free Built-in Control Scanner. See also Connecting to Existing CommandBar Controls.

Then you add a button and set their properties in the way described in Step #5 – Adding a New Toolbar Button. Pay attention to the BeforeID property of the button. To place the button before the New button, you set this property to 3, which is the ID of the button *New*. Please remember that showing an image for any command bar control requires choosing a correct value for the Style property of the button. For the newly added menu item

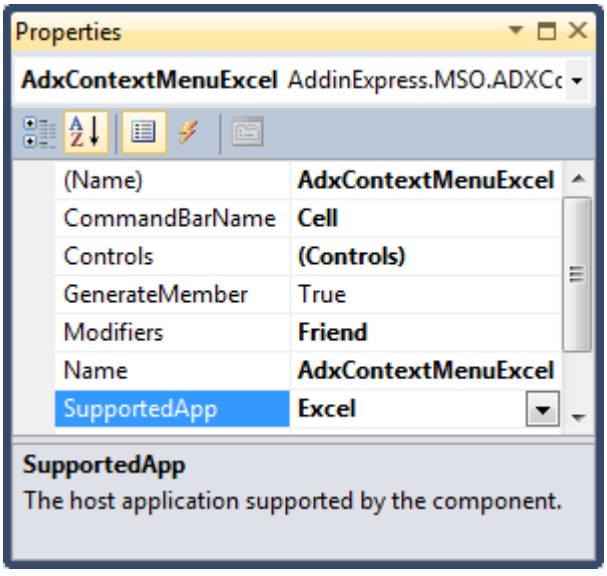
(button) set Style = adxMsoButtonIconAndCaption.



Note that Office imposes restrictions on controls that can be added onto a main menu: the only control types available are command bar button and command bar popup.

See also Step #11 – Customizing the Ribbon User Interface for customizing the Office button menu in Office 2007 and the File tab in Office 2010.

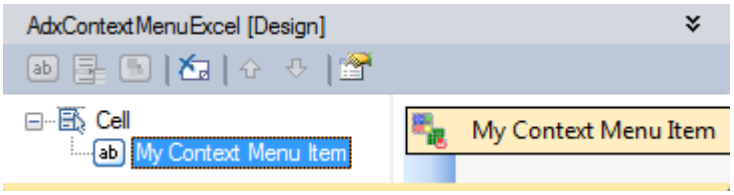
Step #8 - Customizing Context Menus



Add-in Express allows customizing commandbar-based context menus in Office 2000-2010 with the ADXContextMenu component. Its use is similar to that of the ADXMainMenu component. See how to set up such a component to add a custom button to the *Cell* context menu in Excel:

- Add a context menu component to the add-in module
- Specify the host application, the context menu of which you need to customize
- Specify the context menu to customize

- Use the in-place designer to add custom controls to the Controls collection of the component

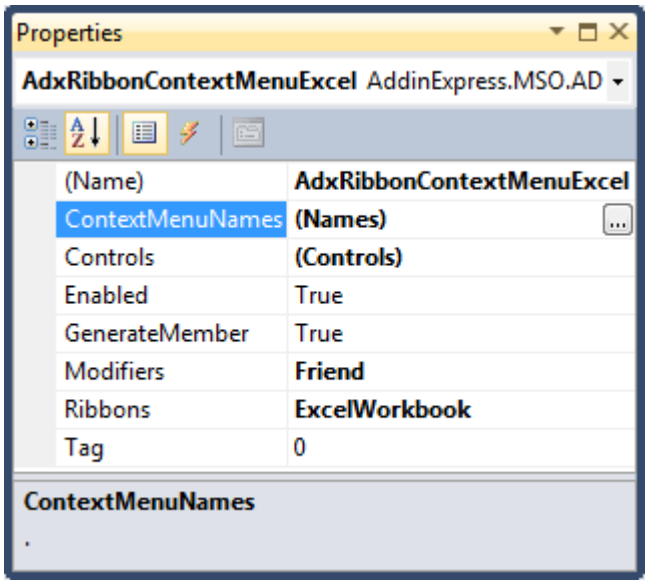


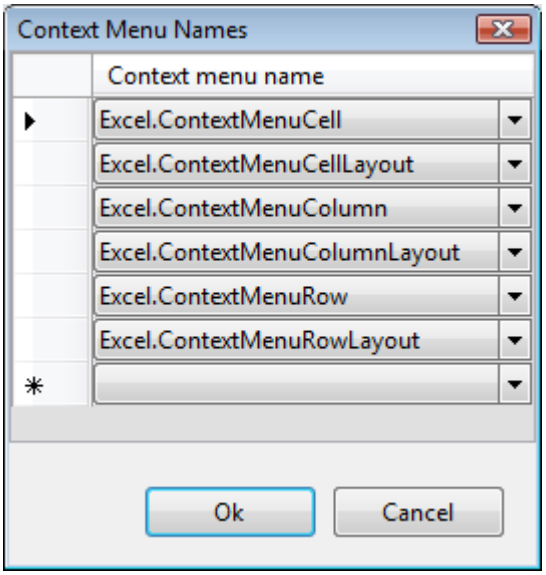
You may want to use the BeforeAddControls event provided by the component to modify the context menu depending on the current context. Say, custom controls in the context menu may reflect the content of an Excel cell, the current chapter of the Word document, etc.

There are several issues related to using command bar based context menus:

- Excel contains two different context menus named Cell. This fact breaks down the command bar development model because the only way to recognize two command bars is to compare their names. This isn't the only exception: see the Built-in Control Scanner to find a number of examples. In this case, the context menu component cannot distinguish context menus. Accordingly, it connects to the first context menu of the name specified by you.
- Command bar based context menu items cannot be positioned in the Ribbon-based context menus: a custom context menu item created with the ADXContextMenu component will always be shown below the built-in and custom context menu items in a Ribbon-based context menu of Office 2010.

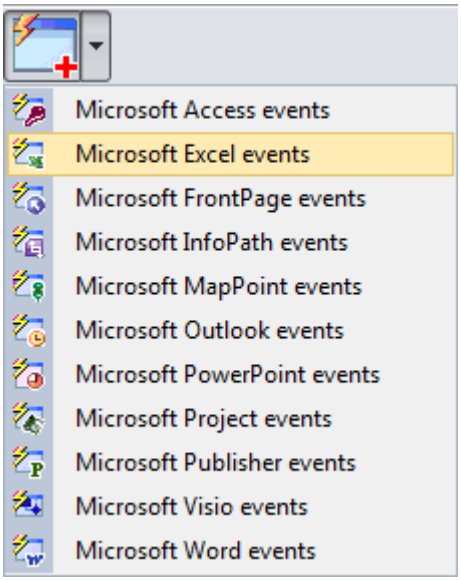
To add a custom item to a context menu in Office 2010, you use the ADXRibbonContextMenu component. Unlike its commandbar-based counterpart (ADXContextMenu), this component allows adding custom Ribbon controls to several context menus in the specified Ribbons. The screenshots below demonstrate component settings required for adding a control to the *ExcelWorkbook* Ribbon. To specify the context menus, to which the control will be added, you use the editor of the ContexMenuNames property of the component.





See also Step #11 – Customizing the Ribbon User Interface.

Step #9 – Handling Host Application Events



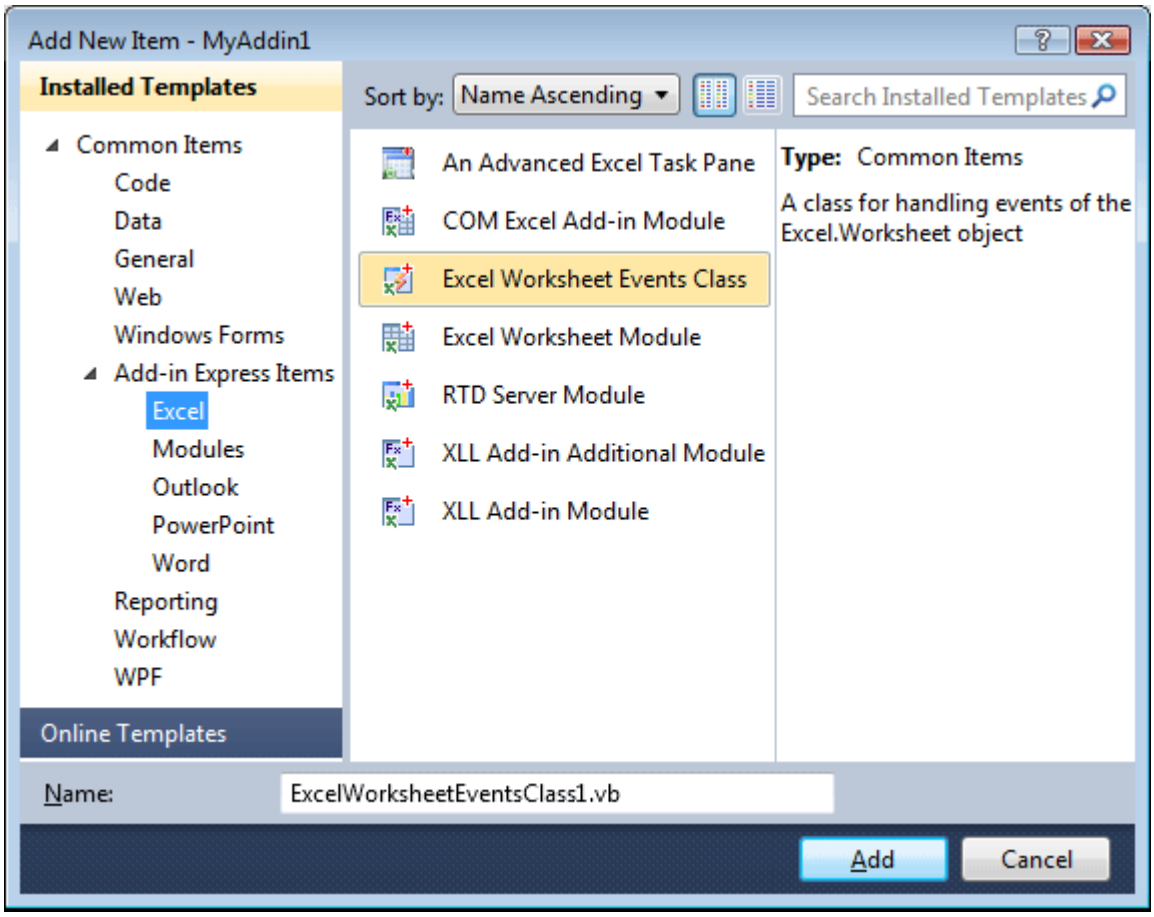
The add-in module designer provides the *Add Events* command that adds (and removes) event components that allow handling application-level events (see Application-level Events).

With the event components, you handle any application-level events of the host application. Say, you may want to disable the button when a window deactivates and enable it when a window activates. The code is as follows:

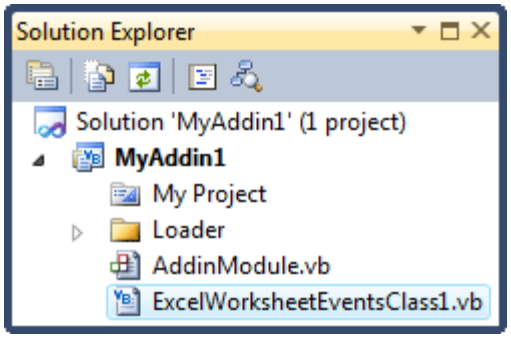
```
Private Sub Deactivate(ByVal sender As Object, _
   ByVal hostObj As Object, ByVal window As Object) _
   Handles adxWordEvents.WindowDeactivate, _
       adxExcelEvents.WindowDeactivate, _
       adxPowerPointEvents.WindowActivate
   Me.AdxCommandBarButton1.Enabled = False
End Sub

Private Sub Activate(ByVal sender As Object, ByVal hostObj As Object, _
   ByVal window As Object) _
   Handles adxWordEvents.WindowActivate, _
       adxExcelEvents.WindowActivate, _
       adxPowerPointEvents.WindowDeactivate
   Me.AdxCommandBarButton1.Enabled = True
End Sub
```

Step #10 – Handling Excel Worksheet Events

Add-in Express provides the Excel Worksheet Events template item (see Add New Item dialog) that allows implementing a set of business rules for an Excel worksheet by handling its events.



This adds an event class to your project.



In the event class, you add the following code to the procedure that handles the BeforeRightClick event of the Worksheet class:

```
Public Overrides Sub ProcessBeforeRightClick(ByVal Target As Object, _
      ByVal E As AddinExpress.MSO.ADXCancelEventArgs)
   Dim R As Excel.Range = CType(Target, Excel.Range)
   'Cancel right-clicks for the first column only
   If R.Address(False, False).IndexOf("A") = 0 Then
      MsgBox("The context menu will not be shown!")
```

```
      E.Cancel = True
   Else
      E.Cancel = False
   End If
End Sub
```

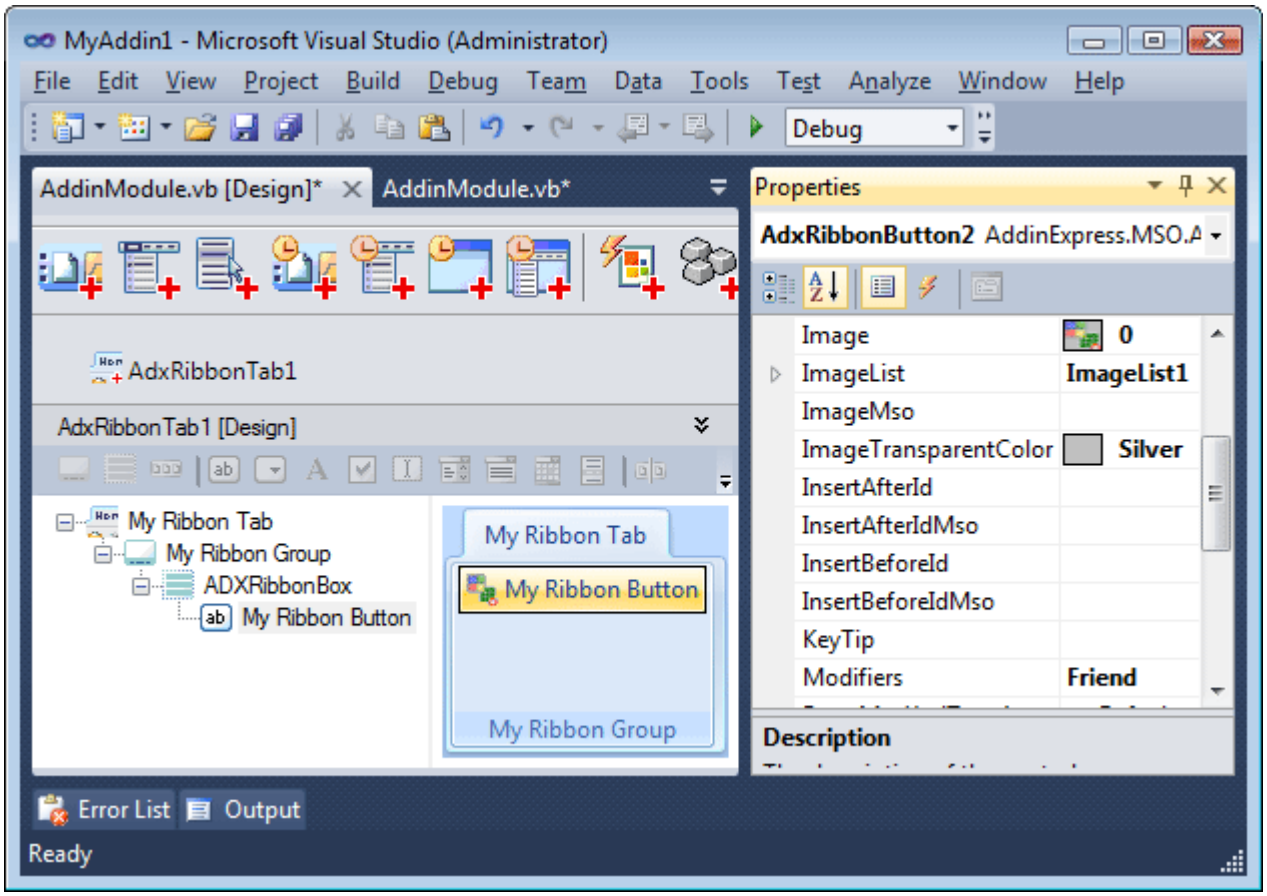
In addition, you modify the Activate and Deactivate procedures as follows:

```
Dim MyEventClass As ExcelWorksheetEventsClass1 = _
       New ExcelWorksheetEventsClass1(Me)
...
Private Sub Deactivate(ByVal sender As Object, ByVal hostObj As Object, _
   ByVal window As Object) _
   Handles adxWordEvents.WindowDeactivate, adxExcelEvents.WindowDeactivate
   Me.AdxCommandBarButton1.Enabled = False
   Select Case Me.HostName
      Case "Excel"
         If MyEventClass.IsConnected Then MyEventClass.RemoveConnection()
      Case "Word"
      Case "PowerPoint"
      Case Else
         MsgBox(Me.AddinName + " doesn't support " + Me.HostName)
   End Select
End Sub

Private Sub Activate(ByVal sender As Object, ByVal hostObj As Object, _
   ByVal window As Object) _
   Handles adxWordEvents.WindowActivate, adxExcelEvents.WindowActivate
   Me.AdxCommandBarButton1.Enabled = True
   Select Case Me.HostName
      Case "Excel"
         If MyEventClass.IsConnected Then MyEventClass.RemoveConnection()
         MyEventClass.ConnectTo(Me.ExcelApp.ActiveSheet, True)
      Case "Word"
      Case "PowerPoint"
      Case Else
         MsgBox(Me.AddinName + " doesn't support " + Me.HostName)
   End Select
End Sub
```

Step #11 – Customizing the Ribbon User Interface

To add a new tab to the Ribbon, you use the *Add Ribbon Tab* command that adds an ADXRibbonTab component to the module. In the in-place visual designer, use toolbar buttons or context menu to add or delete Add-in Express components that form the Ribbon interface of your add-in.

In this sample, you change the caption of your tab to *My Ribbon Tab*. Then, you add a Ribbon group, and change its caption to *My Ribbon Group*. Next, you add a button group. Finally, you add a button and set its caption to *My Ribbon Button*. Use the ImageList and Image properties to set the image for the button.



Now add the event handler to the Click event of the button. Write the following code:

```
Private Sub AdxRibbonButton1_OnClick(ByVal sender As System.Object, _
   ByVal control As AddinExpress.MSO.IRibbonControl, _
   ByVal pressed As System.Boolean) Handles AdxRibbonButton1.OnClick
   AdxCommandBarButton1_Click(Nothing)
End Sub
```

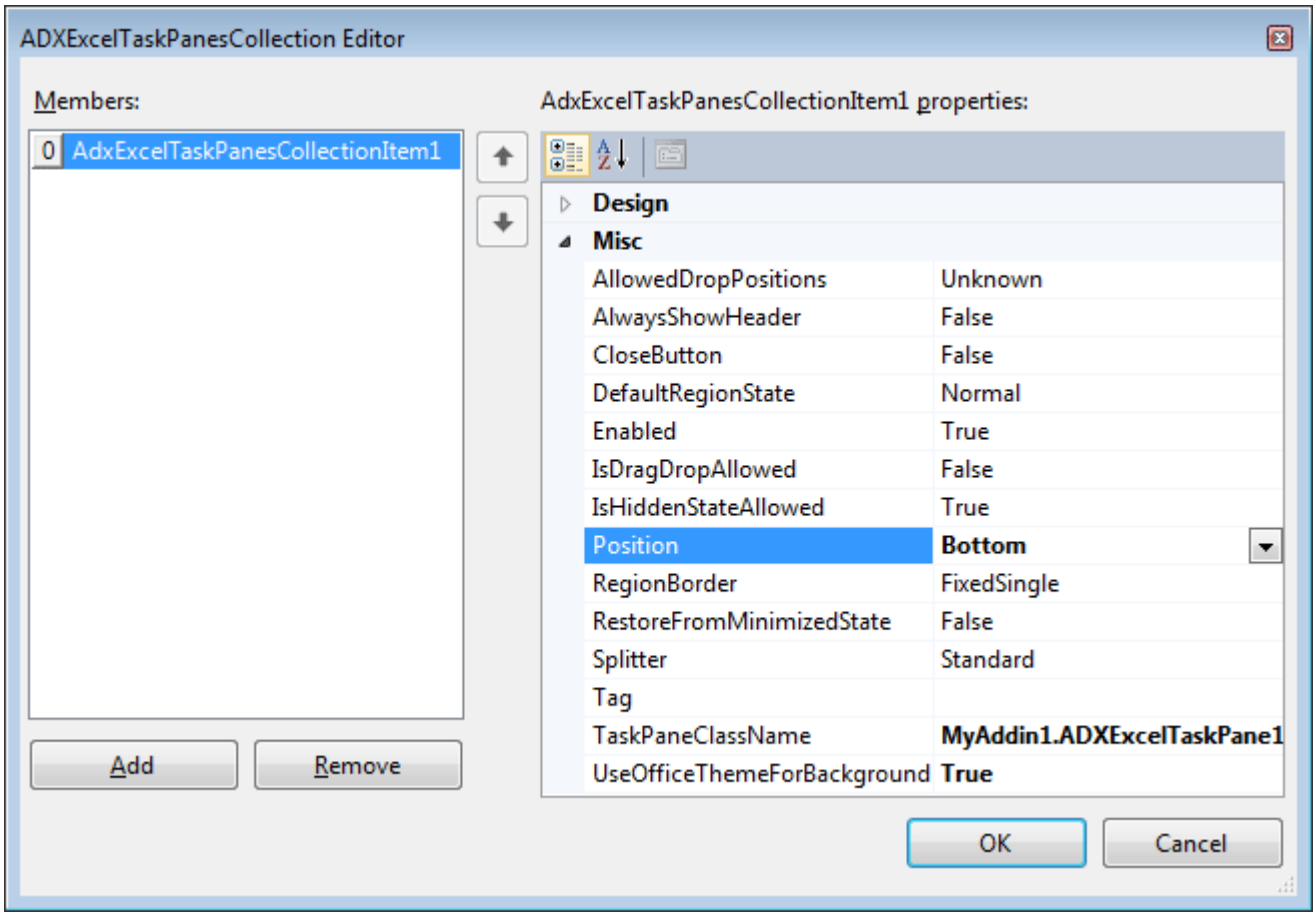
Remember, the Ribbon Tab designer validates the XML-markup automatically, so from time to time you will run into the situation when you cannot add a control to some level. It is a restriction of the Ribbon XML-schema.

In the code of this sample add-in, you can find how you can customize the Office Button menu in Office 2007, see component named AdxRibbonOfficeMenu1. As to the Backstage View, also known as the **File Tab** in Office 2010, the sample project provides the AdxBackstageView1 component that implements the customization shown in Figure 3 at Introduction to the Office 2010 Backstage View for Developers. Note, if you customize the Office Button menu only, Add-in Express maps your controls to the Backstage View when the add-in is loaded by Office 2010. If, however, both Office Button menu and File tab are customized at the same time, Add-in Express ignores custom controls you add to the Office Button menu. See also Ribbon UI.

Step #12 – Adding Custom Task Panes in Excel 2000-2010

Creating a new Excel task pane includes the following steps:

- add an Excel Task Panes Manager (ADXExcelTaskPanesManager) to your add-in module
- add an Add-in Express Excel Task Pane (ADXExcelTaskPane) to your project (see Add New Item dialog)
- add an item to the Items collection of the manager, choose the pane just added in the TaskPaneClassName property of the item and set other properties, such as Position (see the screenshot below).



The properties shown on the screenshot above are:

- AlwaysShowHeader - specifies that the pane header will be shown even if the pane is the only pane in the current region.
- CloseButton - specifies if the Close button will be shown in the pane header. Obviously, there's not much sense in setting this property to true when the header isn't shown.
- Position - specifies the region in which an instance of the pane will be shown. Excel panes are allowed in four regions docked to the four sides of the main Excel window: Right, Bottom, Left, and Top. The fifth region is Unknown.

- TaskPaneClassName - specifies the class name of the Excel task pane.

Now you add a label onto the form and change its caption in the following code:

```
Private Sub RefreshTaskPane()
    Select Case Me.HostName
        Case "Excel"
            Dim Pane As ADXExcelTaskPane1 = _
                TryCast(Me.AdxExcelTaskPanesCollectionItem1.TaskPaneInstance, _
                    ADXExcelTaskPane1)
            If Pane IsNot Nothing Then
                Pane.Label1.Text = Me.GetInfoString()
            End If
        Case "Word"
        Case "PowerPoint"
        Case Else
            'System.Windows.Forms.MessageBox.Show("Invalid host application!")
    End Select
End Sub
```

See also Advanced Custom Task Panes.

Step #13 – Adding Custom Task Panes in PowerPoint 2000-2010

Now you add a PowerPoint task pane:

- add a PowerPoint Task Panes Manager (ADXPowerPointTaskPanesManager) to your add-in module
- add an Add-in Express PowerPoint Task Pane (ADXPowerPointTaskPane) to your project (see Add New Item dialog)
- in the visual designer available for the Controls collection of the manager, add an item to the collection, bind the pane to the item and specify an appropriate value in the Position.

Now add a label onto the form, write a property that reads and updates the label, and update `RefreshTaskPane` in order to set the property value:

```
Private Sub RefreshTaskPane()
    Select Case Me.HostName
        Case "Excel"
...
        Case "Word"
        Case "PowerPoint"
            Dim Pane As ADXPowerPointTaskPane1 = _
                TryCast( _
                    Me.AdxPowerPointTaskPanesCollectionItem1.TaskPaneInstance, _
```

```
                    ADXPowerPointTaskPane1)
            If Pane IsNot Nothing Then
                Pane.Label1.Text = Me.GetInfoString()
            End If
        Case Else
            'System.Windows.Forms.MessageBox.Show("Invalid host application!")
    End Select
End Sub
```

See also Advanced Custom Task Panes and Excel Task Panes.

Step #14 – Adding Custom Task Panes in Word 2000-2010

You add a Word task pane in the same manner:

- add a Word Task Panes Manager (ADXWordTaskPanesManager) to your add-in module
- add an Add-in Express Word Task Pane (ADXWordTaskPane) to your project (see Add New Item dialog)
- in the visual designer available for the Controls collection of the manager, add an item to the collection, bind the pane to the item and specify an appropriate value in the Position.

Now add a label onto the form and update `RefreshTaskPane` in order to set the label:

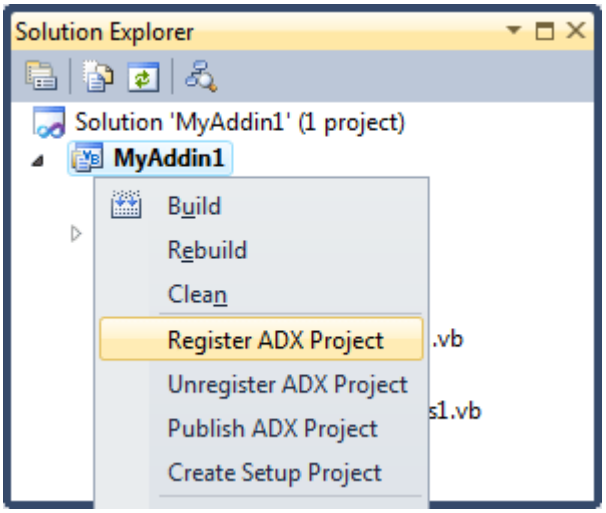
```
Private Sub RefreshTaskPane()
    Select Case Me.HostName
        Case "Excel"
...
        Case "Word"
            Dim Pane As ADXWordTaskPane1 = _
                TryCast( _
                    Me.AdxWordTaskPanesCollectionItem1.CurrentTaskPaneInstance, _
                    ADXWordTaskPane1)
            If Pane IsNot Nothing Then
                Pane.Label1.Text = Me.GetInfoString()
            End If
        Case "PowerPoint"
...
        Case Else
            'System.Windows.Forms.MessageBox.Show("Invalid host application!")
    End Select
End Sub
```

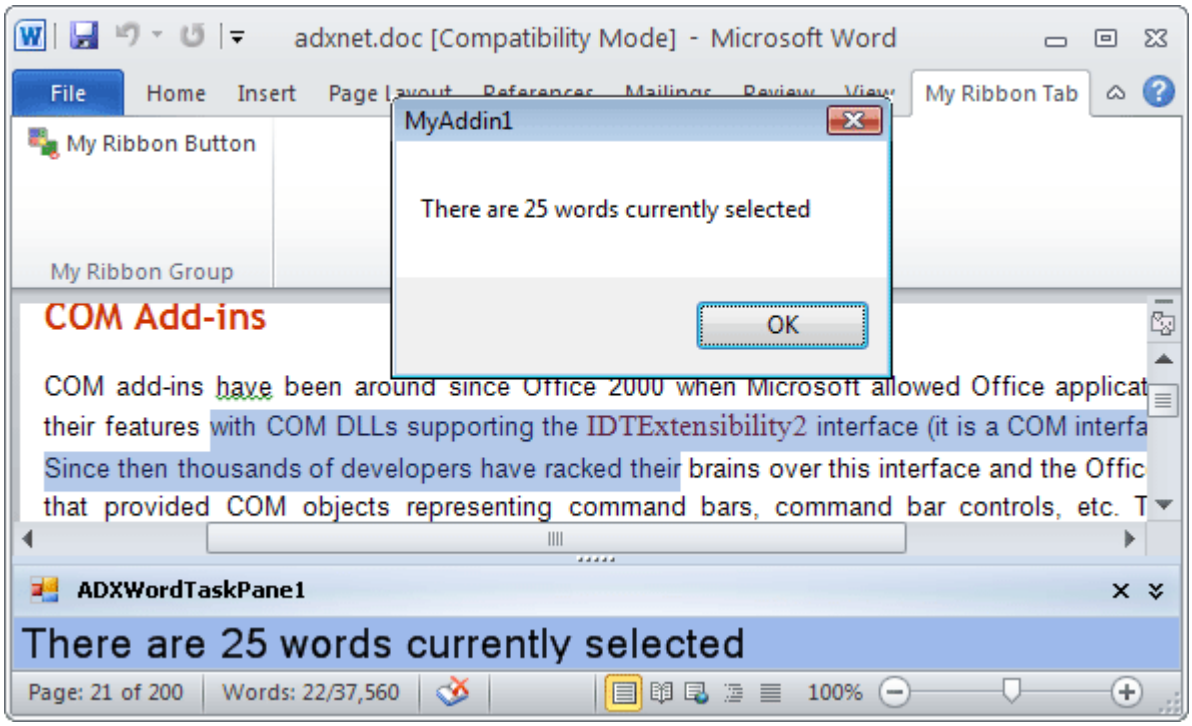
The different names of the properties returning instances of the three pane types reflect the difference in Excel, PowerPoint and Word windowing; while Excel and PowerPoint show their documents in just one main window, Word normally shows documents in multiple windows. In this situation, the Word Task Panes Manager creates one instance of the pane for every document open in Word. Therefore, you need to handle the task pane

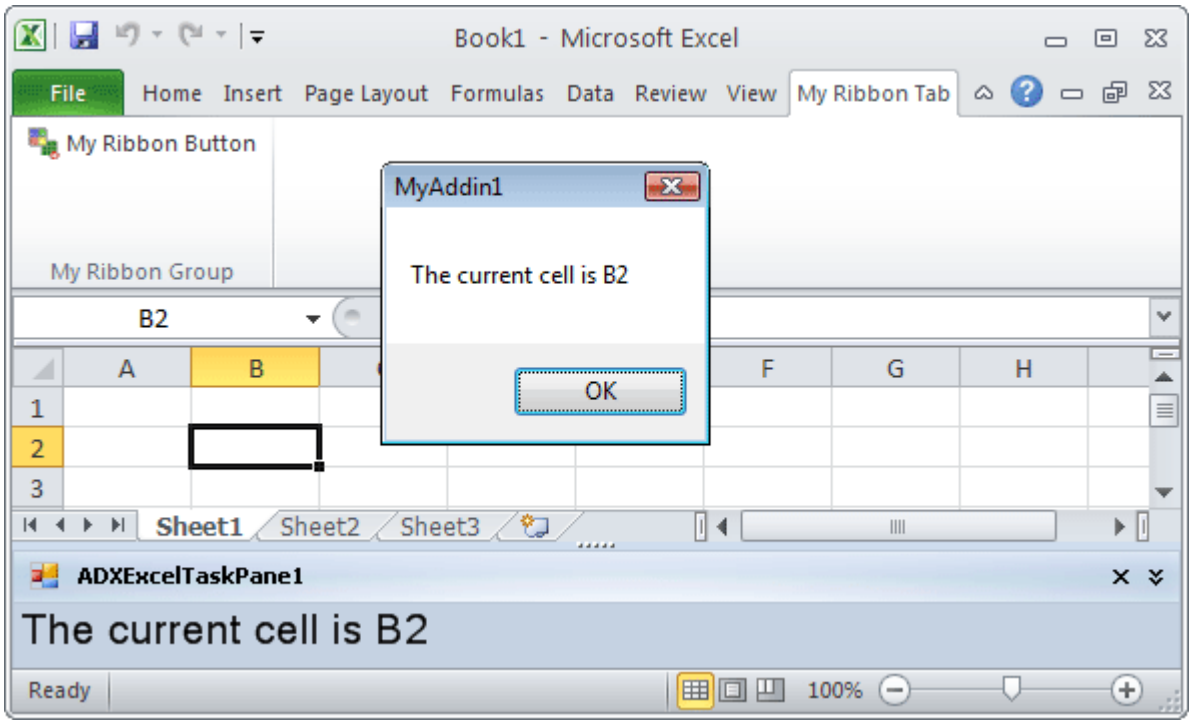
instance, which is currently active. For that reason, the property name is CurrentTaskPaneInstance. See also Advanced Custom Task Panes.

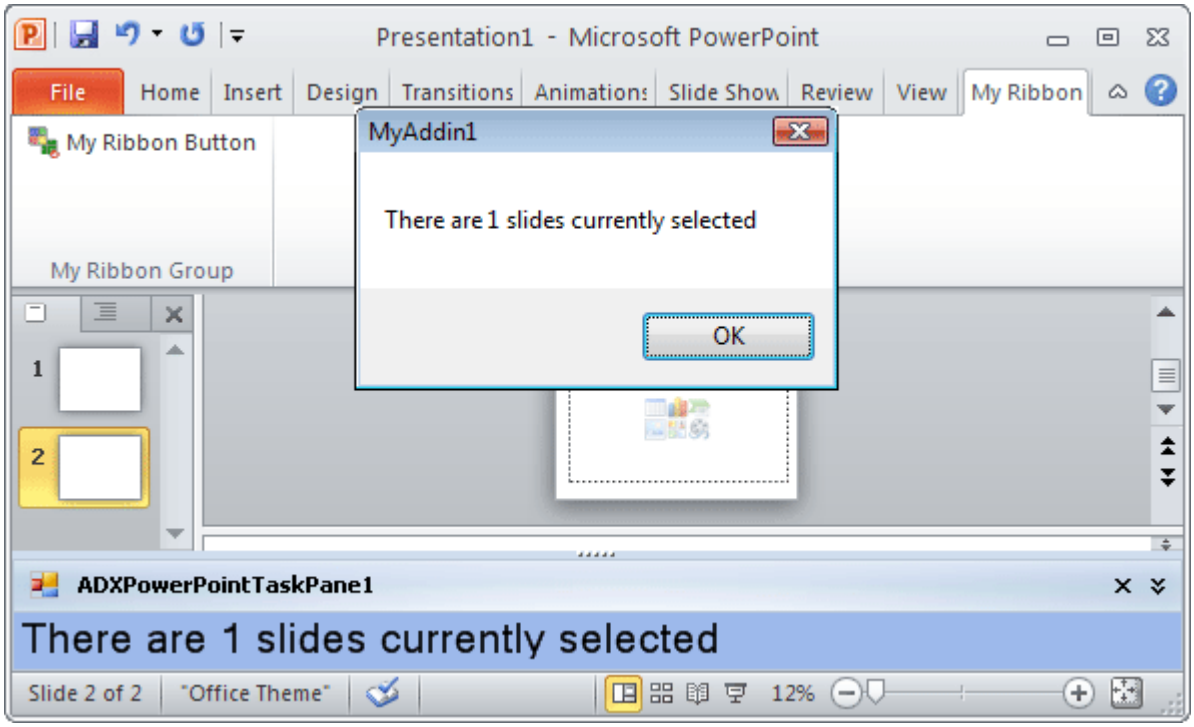
Step #15 – Running the COM Add-in

Choose *Register Add-in Express Project* in the *Build* menu (if you use the Express edition of Visual Studio, this item can be found in the context menu of the add-in module's designer surface), and restart the host applications.





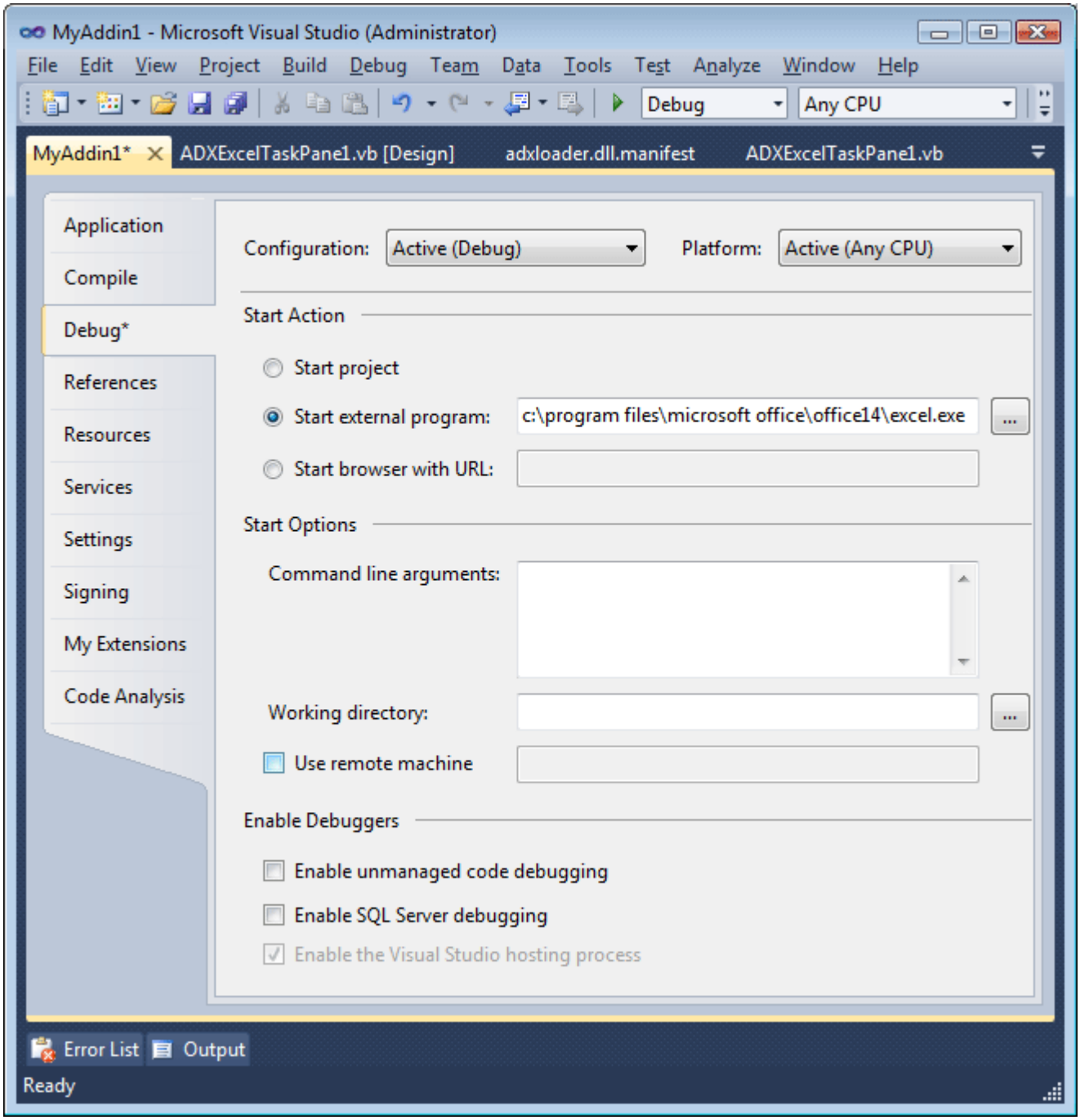




You can also find your add-in in the COM Add-ins Dialog.

Step #16 – Debugging the COM Add-in

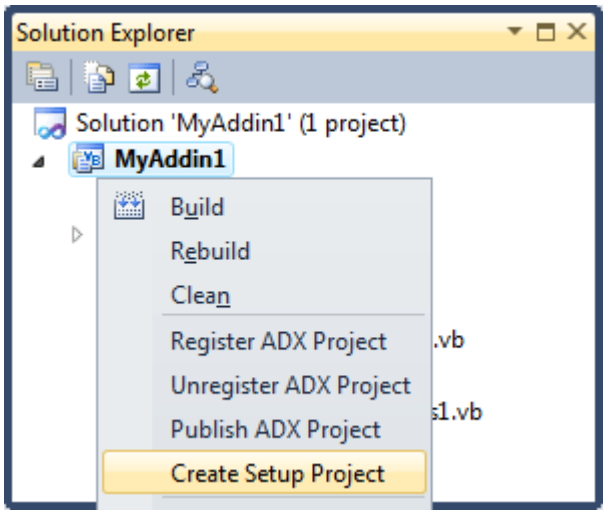
To debug your add-in, in the Project Options window, specify the path to the host application of the add-in in Start External Program and run the project.



Step #17 – Deploying the COM Add-in

Create a setup project, build it, copy all setup files to the target PC and run the installer (see Deploying Office Extensions).

The Deploying Add-in Express Projects section describes MSI-based and ClickOnce Deployment. You can also find some useful tips in the Debugging and Deploying section.

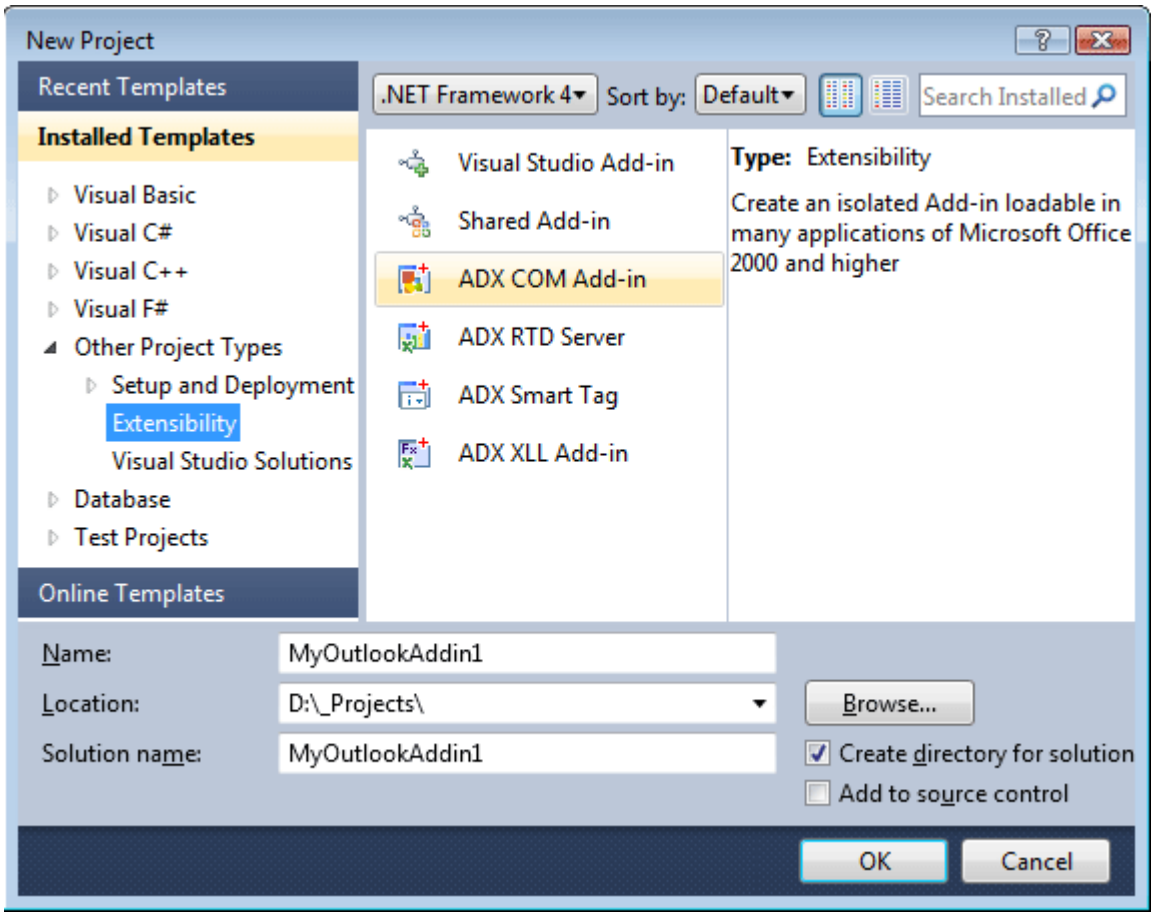


Your First Microsoft Outlook COM Add-in

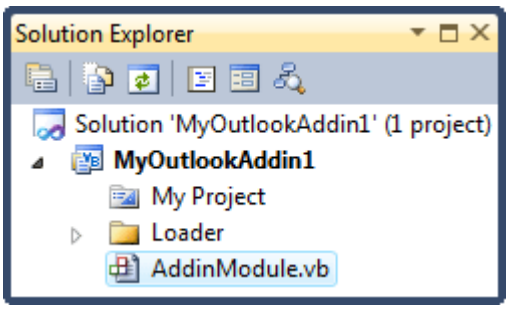
This VB.NET sample shows a project implementing an Outlook COM add-in with the Add-in Express loader as a shim. To understand shims and the Add-in Express loader, see How Your Office Extension Loads Into an Office Application.

Step #1 - Creating an Add-in Express COM Add-in Project

Choose the *Add-in Express COM Add-in* project template in the Visual Studio IDE.



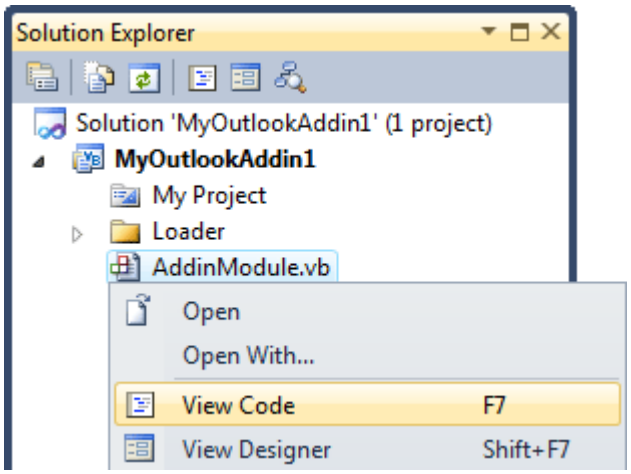
Click OK to start the COM add-in project wizard. In the wizard, you choose the programming language as well as interop assemblies to use and Office applications to support in your add-in.



The project wizard creates and opens a new solution in the IDE. The solution includes the COM add-in project containing the AddinModule.vb (or AddinModule1.cs) file discussed in the next step.

Step #2 - Add-in Module

The AddinModule.vb (or AddinModule1.cs) is a COM add-in module that is the core part of the COM add-in project (see COM Add-ins). It is the container for Add-in Express components, which allow you to concentrate on the functionality of your add-in. You specify the add-in properties in the module's properties, add components to the module's designer, and write the functional code of your add-in in this module. To review its source code, right-click AddinModule1.vb (or AddinModule1.cs) in *Solution Explorer* and choose *View Code* in the context menu.

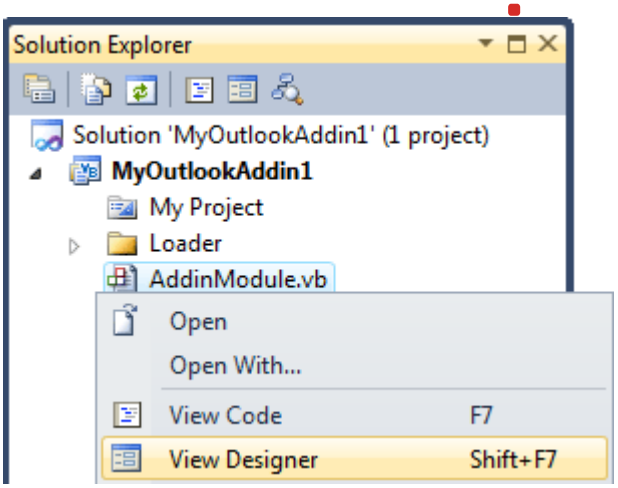


The code for AddinModule1.vb is as follows:

```
Imports System.Runtime.InteropServices
Imports System.ComponentModel

'Add-in Express Add-in Module
<GuidAttribute("3BDF26A5-74E4-42CB-A93A-E88435BC0AD3"), _
      ProgIdAttribute("MyOutlookAddin1.AddinModule")> _
Public Class AddinModule
   Inherits AddinExpress.MSO.ADXAddinModule


#Region " Component Designer generated code. "
   'Required by designer
   Private components As System.ComponentModel.IContainer

   'Required by designer - do not modify
   'the following method
   Private Sub InitializeComponent()
      Me.components = New System.ComponentModel.Container
      '
      'AddinModule
      '
      Me.AddinName = "MyOutlookAddin1"
      Me.SupportedApps = AddinExpress.MSO.ADXOfficeHostApp.ohaOutlook

   End Sub

#End Region

#Region " Add-in Express automatic code "
```

```
    'Required by Add-in Express - do not modify
    'the methods within this region

    Public Overrides Function GetContainer() As _
          System.ComponentModel.IContainer
       If components Is Nothing Then
          components = New System.ComponentModel.Container
       End If
       GetContainer = components
    End Function

    <ComRegisterFunctionAttribute()> _
    Public Shared Sub AddinRegister(ByVal t As Type)
       AddinExpress.MSO.ADXAddinModule.ADXRegister(t)
    End Sub

    <ComUnregisterFunctionAttribute()> _
    Public Shared Sub AddinUnregister(ByVal t As Type)
       AddinExpress.MSO.ADXAddinModule.ADXUnregister(t)
    End Sub

    Public Overrides Sub UninstallControls()
       MyBase.UninstallControls()
    End Sub

#End Region

    Public Sub New()
       MyBase.New()

       'This call is required by the Component Designer
       InitializeComponent()

       'Add any initialization after the InitializeComponent() call

    End Sub

    Public ReadOnly Property OutlookApp() As Outlook._Application
       Get
          Return CType(HostApplication, Outlook._Application)
       End Get
    End Property

End Class
```

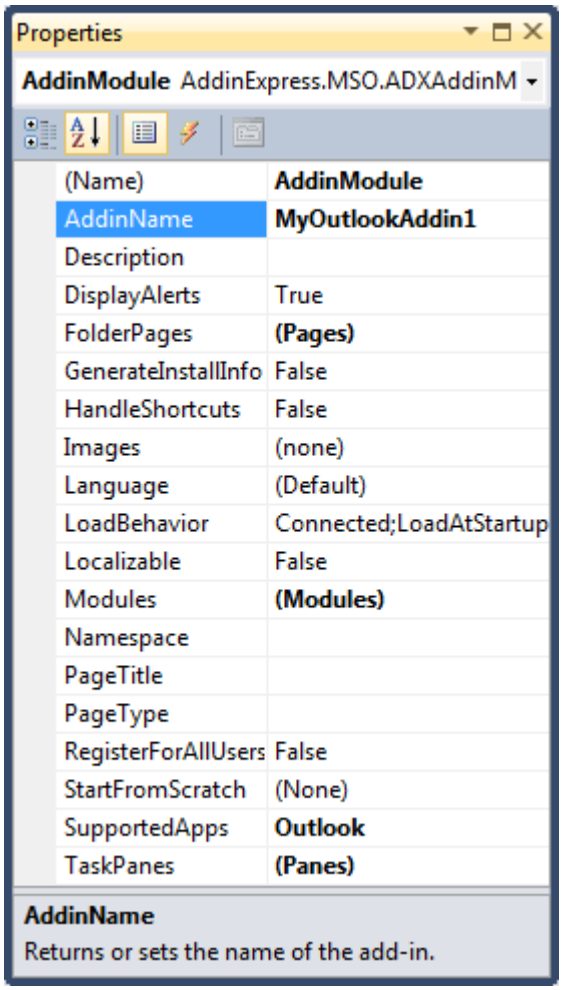
Pay attention to the OutlookApp property of the module generated by the wizard. You use it in your code to get access to Outlook objects, see Step #11 – Accessing Outlook Objects.



Step #3 – Add-in Module Designer

The module designer allows setting add-in properties and adding components to the module. In *Solution Explorer*, right-click AddinModule.vb (or AddinModule.cs) and choose *View Designer* in the context menu.

In the *Properties* window, you specify the name and description of your add-in (see the screenshot below).

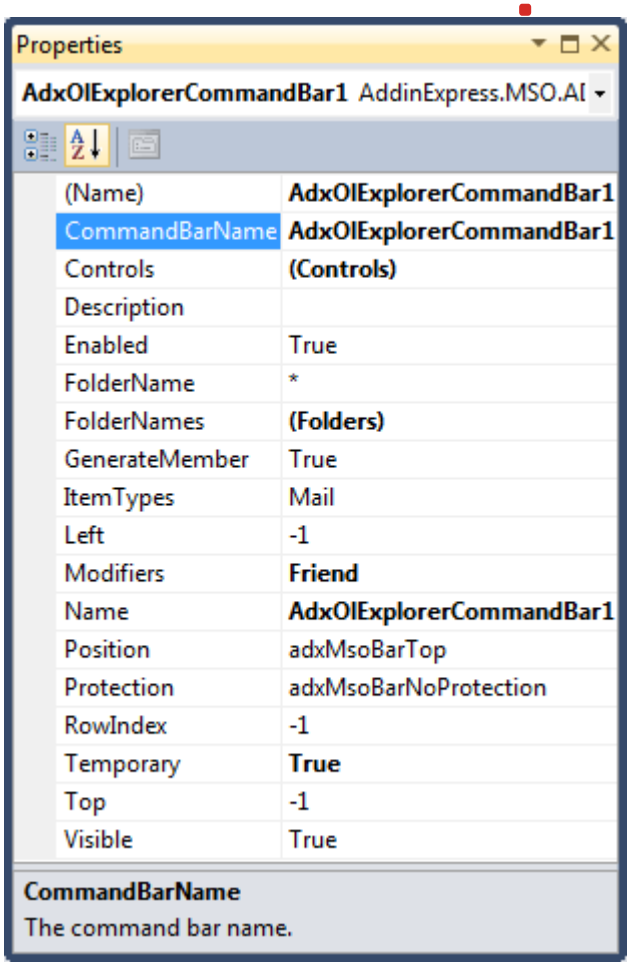


To add an Add-in Express component to the module, you use an appropriate command in the toolbar shown by the add-in module designer. See also Commands of the Add-in Module.

Step #4 – Adding a New Explorer Command Bar

To add a command bar to the Outlook 2000-2007 Explorer window, use the *Add ADXOlExplorerCommandBar* command that adds an ADXOlExplorerCommandBar component to the add-in module.

Select the just added command bar component and, in the *Properties* window, specify the command bar name using the CommandBarName property and choose its position (see the Position property). The component provides several context-sensitive properties: they are FolderName, FolderNames, and ItemTypes (see Outlook CommandBar Visibility Rules).

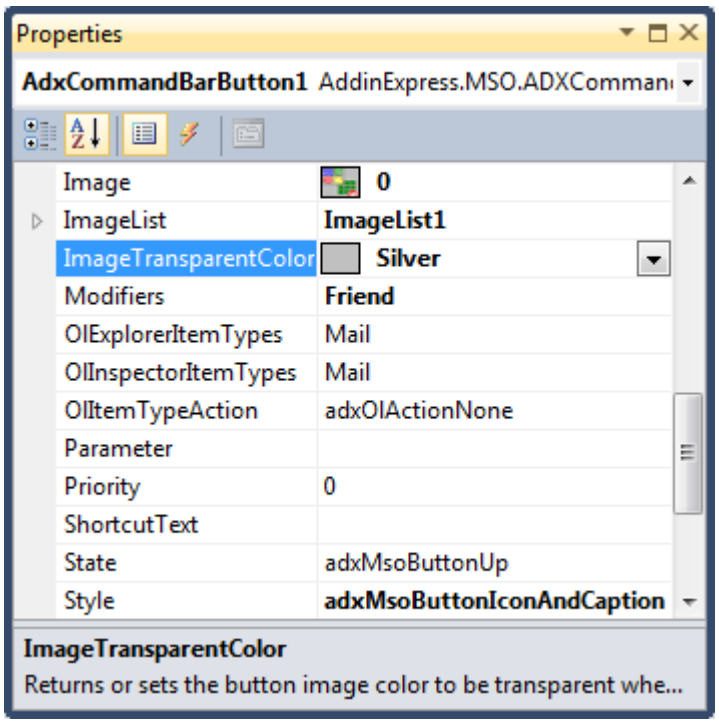


In the screenshot, you see the properties of the Outlook Explorer command bar that will be shown for every Outlook folder (FolderName = "*") the default item types of which are *Mail* or *Task* (see COM Add-ins for Outlook – Template Characters in FolderName).

See also Command Bar UI.

Step #5 – Adding a New Command Bar Button

Select the command bar component on the designer of the add-in module and open the In-place designer area. In this area, you'll see the visual designer of the ADXOlExplorerCommandBar component. Use its toolbar to add or remove command bar controls. Just click the appropriate button and see the result.



To add an icon to the button, add an ImageList to the add-in module and specify the ImageList, Image, and ImageTransparentColor properties of the button. Finally, set the Style property because its default value doesn't show the button image. The screenshot below demonstrates button properties that make the image used in the sample project show up as transparent.

See also Step #11 – Accessing Outlook Objects.

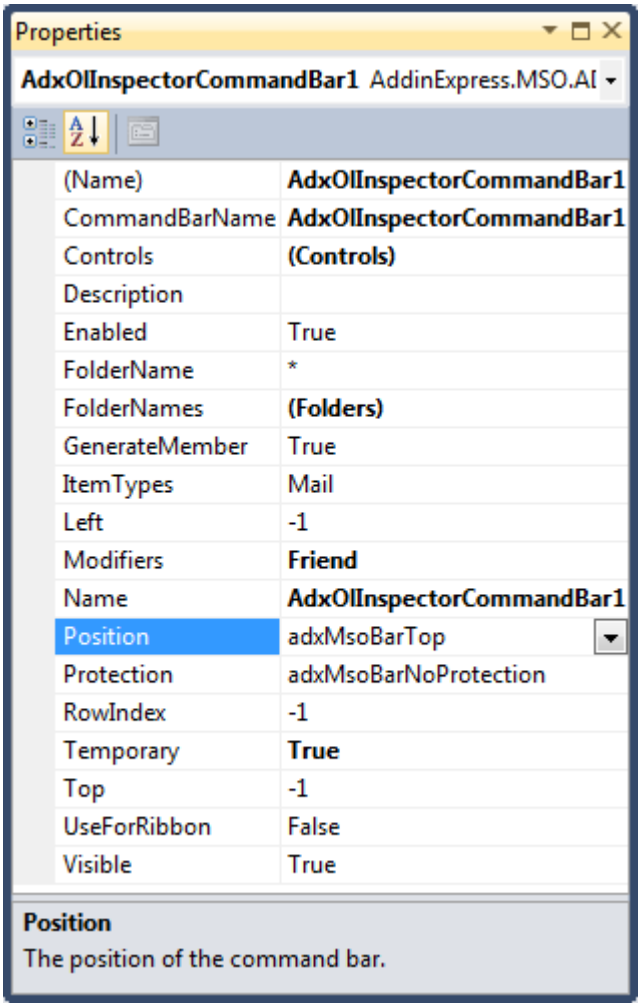
Step #6 – Customizing the Outlook Ribbon User Interface

To add a new custom tab to the Ribbon UI in Outlook 2007-2010, you use the *Add ADXRibbonTab* command that adds an ADXRibbonTab component to the module. The ribbons in which that tab will be shown are set by the Ribbons property. For an Outlook add-in, the default value of this property is "OutlookMailRead;OutlookMailCompose" which means that the tab will be shown in the Mail Inspector windows of Outlook 2007 and higher. In order to show that tab in the Outlook 2010 Explorer windows too, set the Ribbons property to "OutlookMailRead;OutlookMailCompose;OutlookExplorer".

In the visual designer for the ADXRibbonTab component, you populate the tab with Add-in Express components that form the Ribbon interface of your add-in. First, you add a Ribbon tab and change its caption to *My Ribbon Tab*. Then you select the tab component, add a Ribbon group, and change its caption to *My Ribbon Group*. Next, you select the group, and add a button group. Finally, you select the button group and add a button. Set the button caption to *My Ribbon Button*. Use the ImageList, Image, and ImageTransparentColor properties to set the icon for the button. See also Ribbon UI.

Step #7 – Adding a New Inspector Command Bar

To add a command bar to the Outlook 2000-2003 Inspector windows, use the *Add ADXOlInspectorCommandBar* command that adds an ADXOlInspectorCommandBar component to the add-in module.



The Inspector command bar component provides the same properties as the Explorer command bar component. In the screenshot, you see the default values for the Inspector command bar.

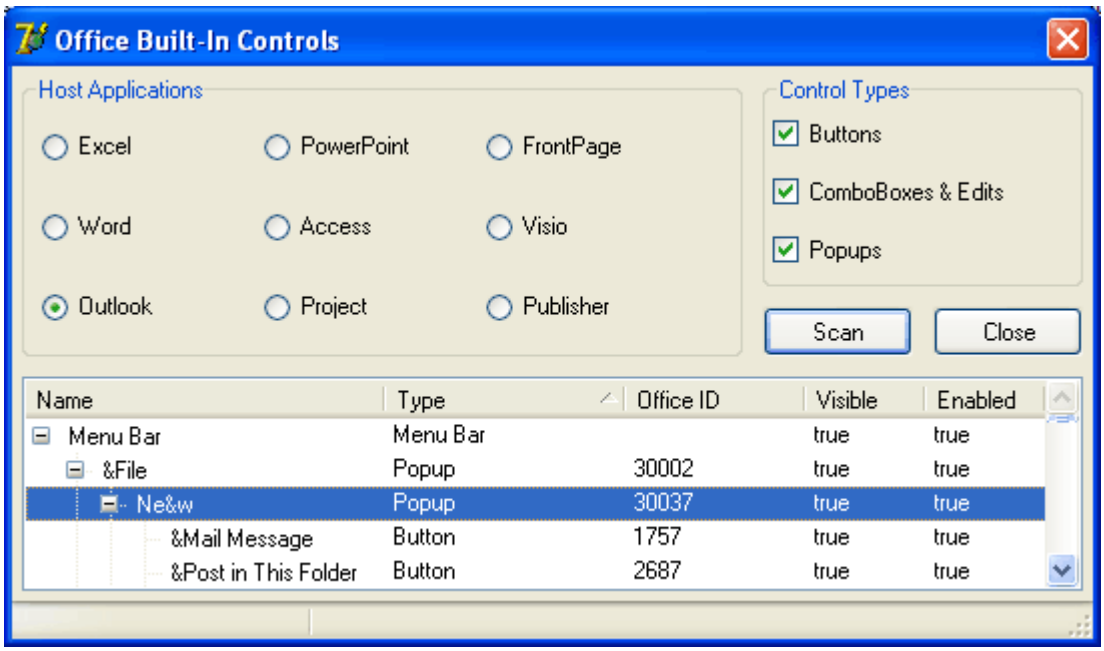
If you specify the full path to a folder in the FolderName (FolderNames) property of an inspector command bar component, the corresponding toolbar is displayed for inspectors that open Outlook items the Parent properties of which point to that folder.

For adding a new command bar button onto the inspector toolbar see Step #5 – Adding a New Command Bar Button. See also Step #11 – Accessing Outlook Objects, Command Bar UI, Outlook CommandBar Visibility Rules, COM Add-ins for Outlook – Template Characters in FolderName, and Releasing COM objects.

Step #8 – Customizing Main Menu in Outlook 2000-2007

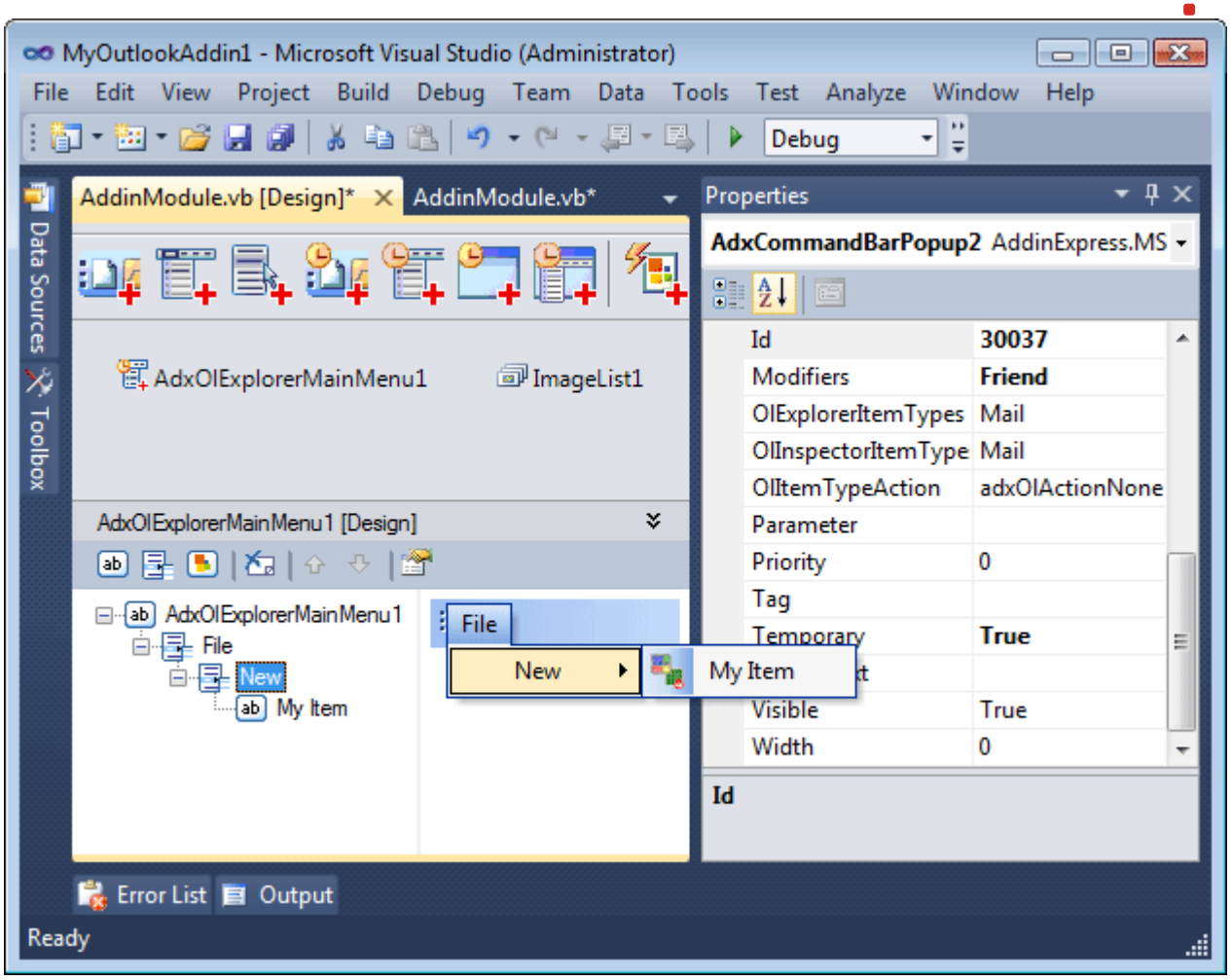
Outlook 2000-2003 provides two main menu types. They are available for two main types of Outlook windows: Explorer and Inspector. Accordingly, Add-in Express provides two main menu components: Explorer Main Menu component and Inspector Main Menu component (note the Ribbon UI replaces the main menu of Inspector windows in Outlook 2007 and all main menus in Outlook 2010). You add either of them using the Commands toolbar of the add-in module. Then you use the in-place visual designer of the component to populate it with controls.

For instance, to add a custom control to the popup shown by the *File* | *New* item in Outlook 2000-2007 Explorer windows, you start our free Built-in Control Scanner to scan the command bars and controls of Outlook. The screenshot below shows the result of scanning. You need the Office IDs from the screenshot to bind Add-in Express components to the corresponding controls:



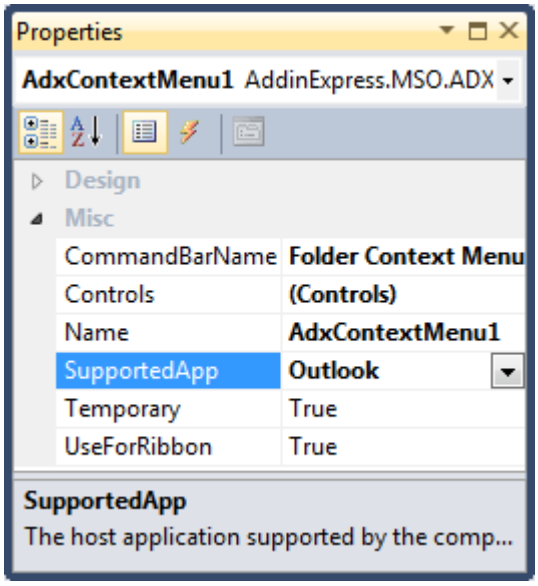
- Add a popup control to the menu component and set its Id property to 30002
- Add a popup control to the popup control above and set its Id to 30037
- Add a button to the popup above and specify its properties. To show your button before the Mail Message button, set its BeforeID property to 1757 (see the screenshot above)

The following screenshot shows the settings of the popup created at step 2 above:



See also Step #11 – Accessing Outlook Objects and Connecting to Existing CommandBar Controls.

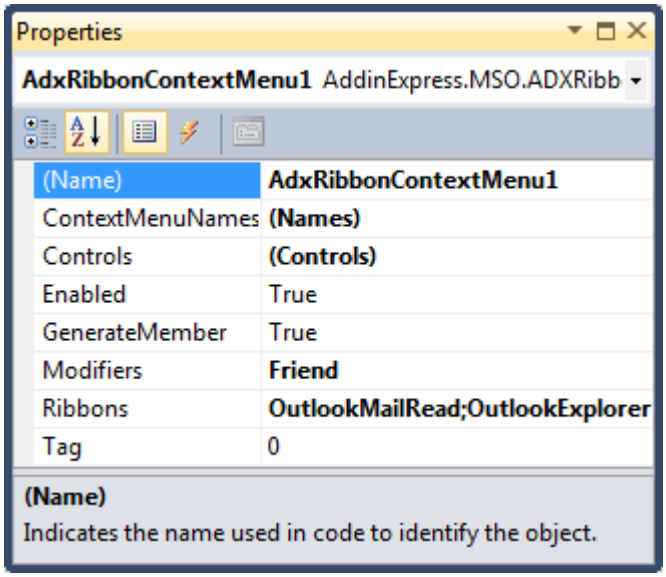
Step #9 - Customizing Context Menus in Outlook

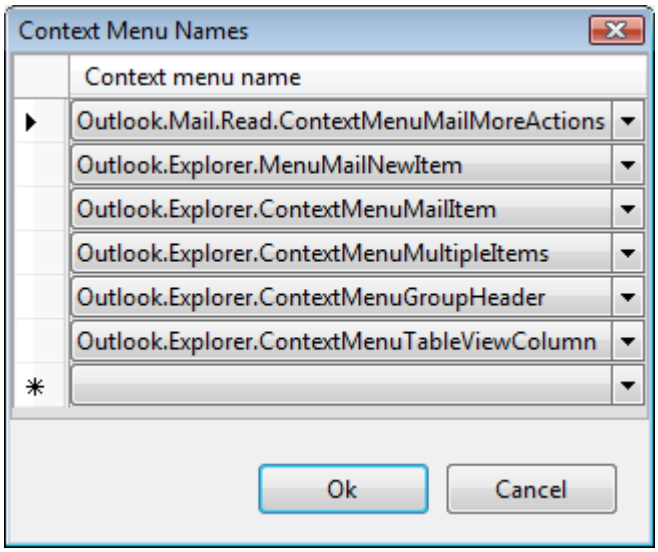


Add-in Express allows customizing commandbar-based context menus of Outlook 2002-2007 via the Context Menu component (Outlook 2000 context menus aren't customizable!). You use the context menu of the add-in module to add such a component onto the module. Then you choose Outlook in the SupportedApp property of the component. Then, in the CommanBarName property, you choose the context menu you want to customize. Finally, you add custom controls in the visual designer supplied for the Controls property.

The sample add-in described in this chapter adds a custom item to the Folder Context Menu command bar that implements the context menu which is shown when you right-click a folder in the folder tree.

Also, you can customize many Ribbon-based context menus in Outlook 2010. The *Add ADXRibbonContextMenu* command of the add-in module adds an ADXRibbonContextMenu component that allows specifying Ribbons that supply context menu names for the ContextMenuNames property. You use the ContextMenuNames property editor to choose the context menu(s) that will display your custom controls specified in the Controls property.



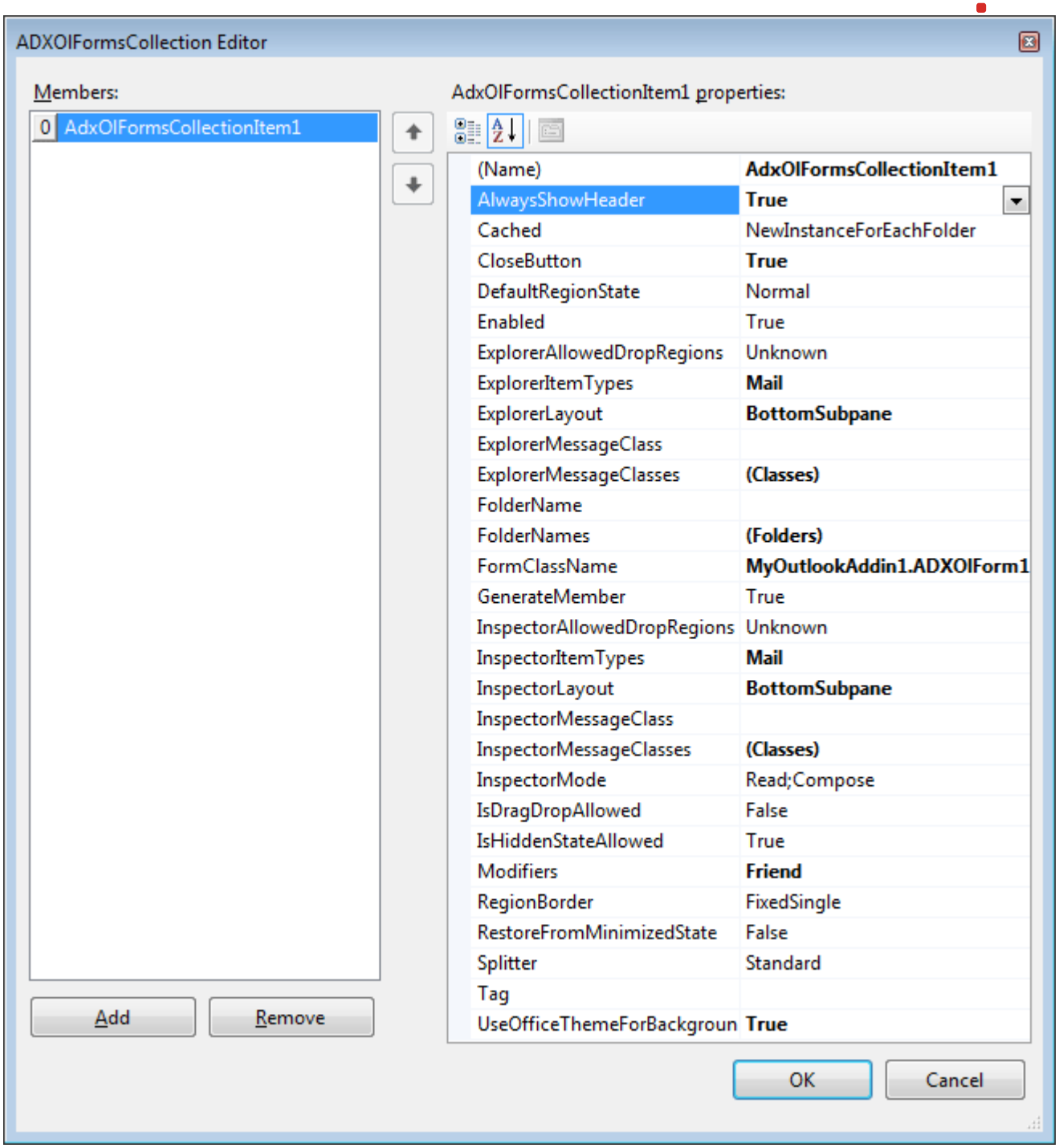


See also Step #11 – Accessing Outlook Objects.

Step #10 - Adding a Custom Task Pane in Outlook 2000-2010

You start with adding an Add-in Express Outlook Form to your project (see Add New Item dialog). Then you add an Outlook Forms Manager component onto your add-in module (see Commands of the Add-in Module). Finally, you add an item to the Items collection of the manager component and set the following properties of the item:

- ExplorerItemTypes = Mail – your form will be shown for all mail folders
- ExplorerLayout = BottomSubpane – the task pane will be shown below the list of mails in Outlook Explorer
- InspectorItemTypes = Mail – an instance of the form will be shown whenever you open an e-mail
- InspectorLayout = BottomSubpane – your task pane will be shown below the message body
- AlwaysShowHeader = True – the header containing the icon (a 16x16 .ico) and the caption of your form will be shown for your form even if it is a single form in the given region
- CloseButton = True – the header will contain the *Close* button; a click on it generates the OnADXBeforeCloseButtonClick event of the form
- FormClassName = MyOutlookAddin1.ADXOlForm1 – the class name of the form



See also Step #12 – Handling Outlook Events, Advanced Custom Task Panes and Advanced Outlook Regions.

Step #11 – Accessing Outlook Objects

Add the following method to the add-in module:

```
Friend Function GetSubject(ByVal InspectorOrExplorer As Object) As String
```

```
    Dim item As Object = Nothing
    Dim selection As Outlook.Selection = Nothing

    If TypeOf InspectorOrExplorer Is Outlook.Explorer Then
        Try
            'Explorer.Selection fires an exception for a top-level folder
            selection = CType(InspectorOrExplorer, Outlook.Explorer).Selection
            item = selection.Item(1)
        Catch
        Finally
            If selection IsNot Nothing Then Marshal.ReleaseComObject(selection)
        End Try
    ElseIf TypeOf InspectorOrExplorer Is Outlook.Inspector Then
        Try
            item = CType(InspectorOrExplorer, Outlook.Inspector).CurrentItem
        Catch
        End Try
    End If

    If item Is Nothing Then
        Return ""
    Else
        Dim subject As String = "The subject is:" + "'" + _
            item.GetType().InvokeMember("Subject", _
                Reflection.BindingFlags.GetProperty, _
                Nothing, item, Nothing).ToString() _
            + "'"
        Marshal.ReleaseComObject(item)
        Return subject
    End If
End Function
```

The code of the GetSubject method emphasizes the following:

- Outlook fires an exception when you try to obtain the Selection object for a top-level folder, such as *Personal Folders*
- There may be no items in the Selection object
- All COM objects created in your code must be released, see Releasing COM objects

Now select the buttons added in previous steps in the *Properties* window combo one by one and create the following event handlers:

```
Private Sub ActionInExplorer(ByVal sender As System.Object) _
   Handles AdxCommandBarButton1.Click
    Dim explorer As Outlook.Explorer = Me.OutlookApp.ActiveExplorer
    If explorer IsNot Nothing Then
```

```
        MsgBox(GetSubject(explorer))
        Marshal.ReleaseComObject(explorer)
    End If
End Sub

Private Sub ActionInInspector(ByVal sender As System.Object) _
         Handles AdxCommandBarButton2.Click, AdxCommandBarButton6.Click
    Dim inspector As Outlook.Inspector = Me.OutlookApp.ActiveInspector
    If inspector IsNot Nothing Then
        MsgBox(GetSubject(inspector))
        Marshal.ReleaseComObject(inspector)
    End If
End Sub

Private Sub AdxRibbonButton1_OnClick(ByVal sender As System.Object, _
    ByVal control As AddinExpress.MSO.IRibbonControl, _
    ByVal pressed As System.Boolean) Handles AdxRibbonButton1.OnClick

    Dim context As Object = control.Context
    If TypeOf context Is Outlook.Inspector Then
        ' Outlook 2007 and higher
        ActionInInspector(Nothing)
    ElseIf TypeOf context Is Outlook.Explorer Then
        ' Outlook 2010 and higher
        ActionInExplorer(Nothing)
    Else
        ' there can be a lot of other contexts in Outlook 2010,
        ' see http://msdn.microsoft.com/en-us/library/ee692172(office.14).aspx
    End If
    Marshal.ReleaseComObject(context)
End Sub
```

Step #12 – Handling Outlook Events

The COM add-in designer provides the *Add Events* command that adds (and removes) event components that allow handling application-level events. In this sample, we add the Outlook Events component to the add-in module.

With the Outlook Events component, you handle any application-level events of Outlook. For instance, the following code handles the BeforeFolderSwitch event of the Outlook Explorer class:

```
Private Sub adxOutlookEvents_ExplorerBeforeFolderSwitch _
   (ByVal sender As Object, _
   ByVal e As AddinExpress.MSO.ADXOlExplorerBeforeFolderSwitchEventArgs) _
   Handles adxOutlookEvents.ExplorerBeforeFolderSwitch
```

```
    MsgBox("You are switching to the " + e.NewFolder.Name + " folder")
End Sub
```

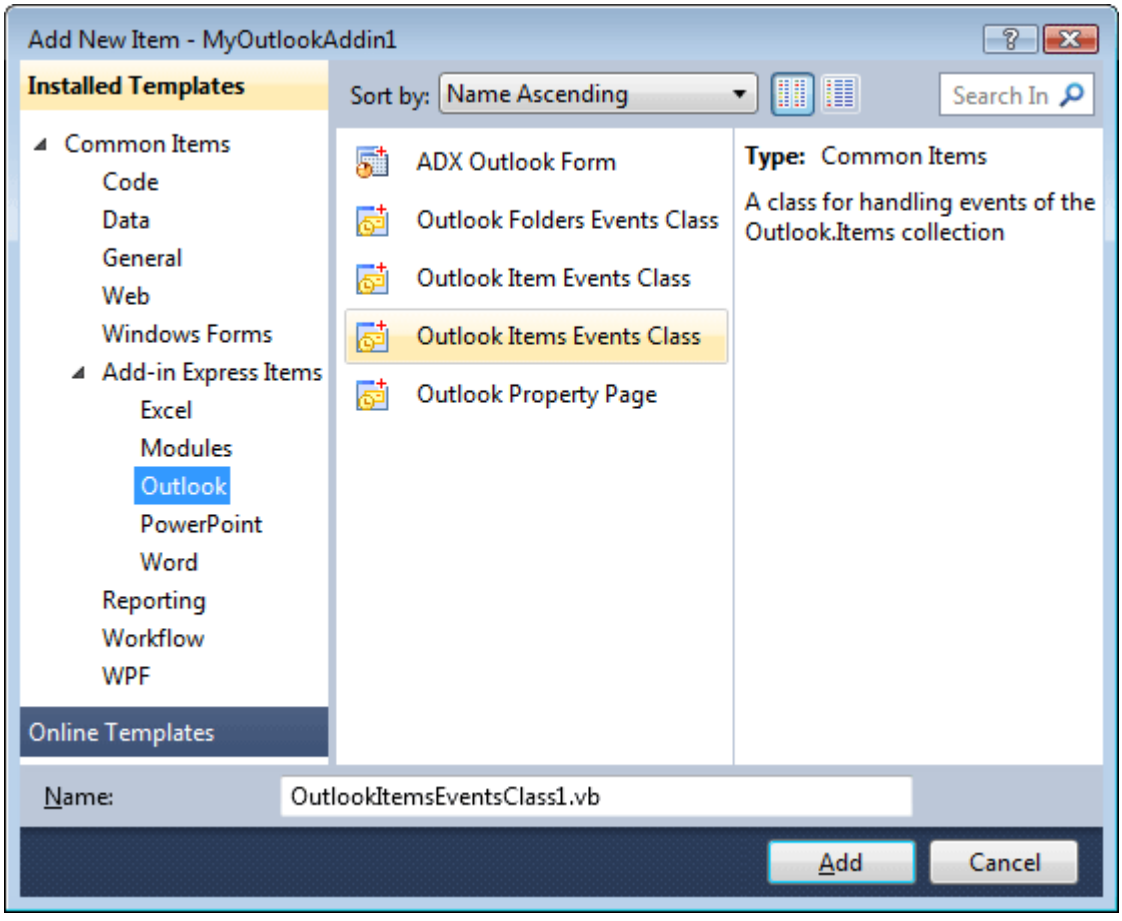
On the form added in Step #10 – Adding a Custom Task Pane in Outlook 2000-2010, you add a label and handle, say the ADXSelectionChange event of the form:

```
Private Sub ADXOlForm1_ADXSelectionChange() Handles MyBase.ADXSelectionChange
    Me.Label1.Text = CType(Me.AddinModule, MyOutlookAddin1.AddinModule) _
        .GetSubject(Me.ExplorerObj)
End Sub
```

See also Step #13 – Handling Events of Outlook Items Object and Events Classes

Step #13 – Handling Events of Outlook Items Object

The Outlook MAPIFolder class provides the Items collection. This collection provides the following events: ItemAdd, ItemChange, and ItemRemove. To process these events, you use the *Outlook Items Events Class* item located in the Add New Item dialog:



This adds the OutlookItemsEventsClass1.vb class to the add-in project. You handle the ItemAdd event by entering some code into the ProcessItemAdd procedure of the class:

```
Imports System

'Add-in Express Outlook Items Events Class
Public Class OutlookItemsEventsClass1
   Inherits AddinExpress.MSO.ADXOutlookItemsEvents

   Public Sub New(ByVal ADXModule As AddinExpress.MSO.ADXAddinModule)
      MyBase.New(ADXModule)
   End Sub

   Public Overrides Sub ProcessItemAdd(ByVal Item As Object)
      MsgBox("The item with subject '" + Item.Subject + _
         "' has been added to the Inbox folder")
   End Sub

   Public Overrides Sub ProcessItemChange(ByVal Item As Object)
      'TODO: Add some code
   End Sub

   Public Overrides Sub ProcessItemRemove()
      'TODO: Add some code
   End Sub
End Class
```

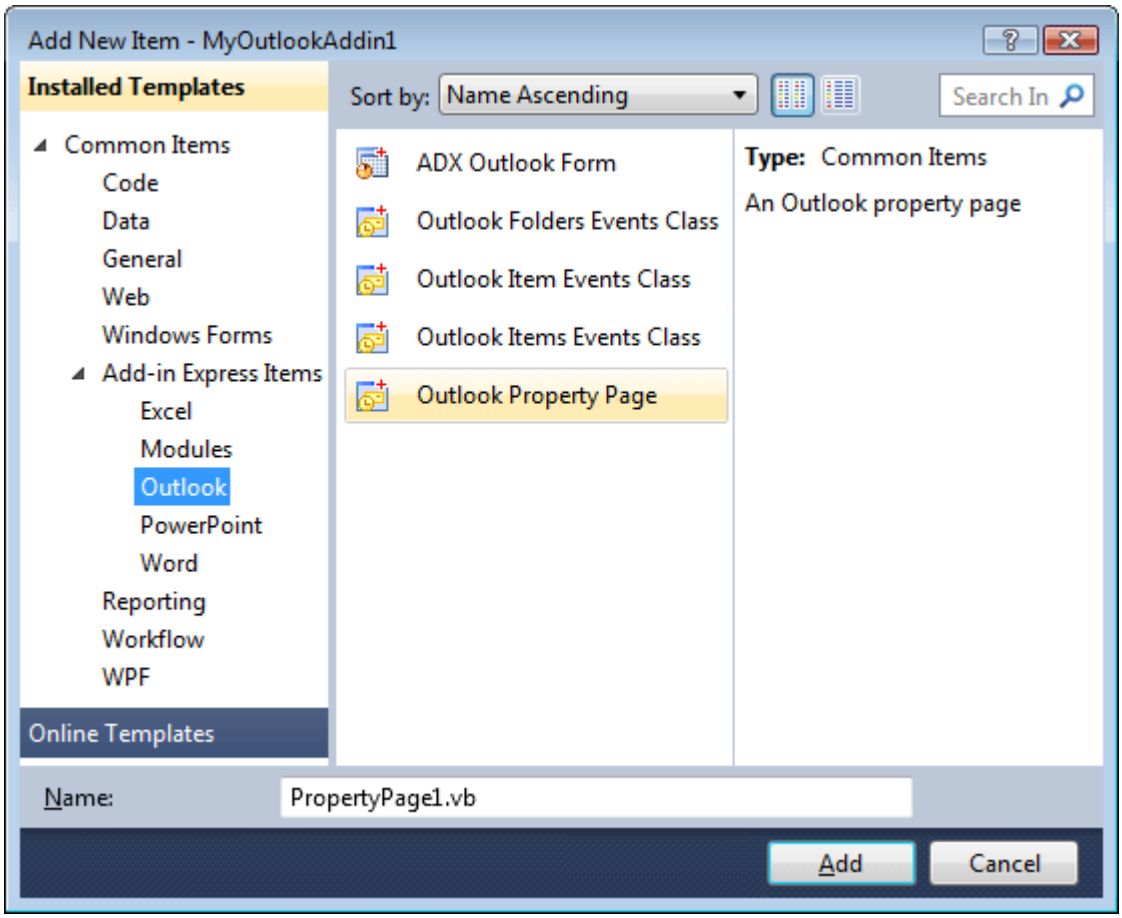
To use this class, you have to add the following declarations and code to the add-in module:

```
Dim ItemsEvents As OutlookItemsEventsClass1 = _
       New OutlookItemsEventsClass1(Me)

Private Sub AddinModule_AddinBeginShutdown(ByVal sender As Object, _
      ByVal e As System.EventArgs) Handles Me.AddinBeginShutdown
   If ItemsEvents IsNot Nothing Then
      ItemsEvents.RemoveConnection()
      ItemsEvents = Nothing
   End If
End Sub

Private Sub AddinModule_AddinStartupComplete(ByVal sender As Object, _
      ByVal e As System.EventArgs) Handles Me.AddinStartupComplete
   ItemsEvents.ConnectTo( _
         AddinExpress.MSO.ADXOlDefaultFolders.olFolderInbox, True)
End Sub
```

To process events of the Folders and Items classes as well as of all item sorts in Outlook, see Events Classes.

Step #14 – Adding Property Pages to the Folder Properties Dialog

Outlook allows adding custom pages (tabs) to the Options dialog (the *Tools* | *Options* menu) and / or to the *Properties* dialog of any folder. To automate this task, Add-in Express provides the ADXOlPropertyPage component. You find it in the Add New Item dialog (see the screenshot below).

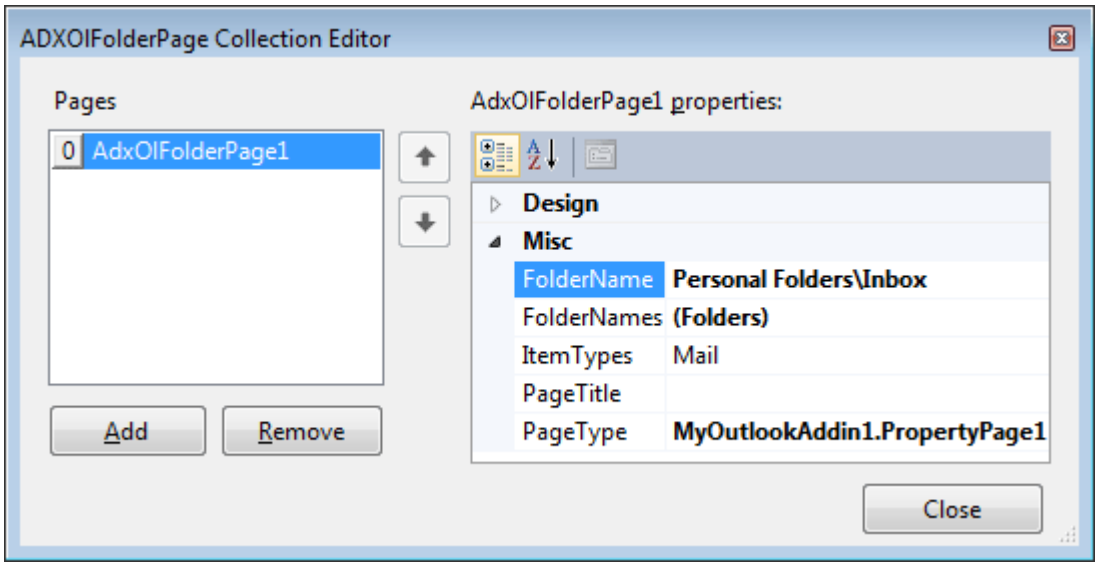


Click the *Add* button to add a descendant of the ADXOlPropertyPage class that implements the IPropertyPage COM interface to your project. You can customize that page as an ordinary form: add the controls and handle their events.

To add a property page to the *<folder name> Properties* dialog box of an Outlook folder(s), you do the following:

- In the add-in module properties, run the editor of the FolderPages property,
- Click the Add button,
- Specify the folder you need in the FolderName property,
- Set the PageType property to the property page component you've added
- Specify the Title property and close the dialog box.

The screenshot below shows the settings you need to display your page in the *Folder Properties* dialog for the *Inbox* folder.



The path to the *Inbox* folder depends on the environment as well as on the Outlook localization. To take care of this, get the path to the *Inbox* folder at add-in startup and assign it to the FolderName property of the *Folder Page* item. This can be done with the following code that handles the AddinStartupComplete event in the add-in module:

```
Private Sub AddinModule_AddinStartupComplete(ByVal sender As Object, _
        ByVal e As System.EventArgs) Handles Me.AddinStartupComplete
   ItemsEvents.ConnectTo( _
      AddinExpress.MSO.ADXOlDefaultFolders.olFolderInbox, True)
   Dim ns As Outlook.NameSpace = Me.OutlookApp.GetNamespace("Mapi")
   Dim folder As Outlook.MAPIFolder = _
      ns.GetDefaultFolder(Outlook.OlDefaultFolders.olFolderInbox)
   Me.FolderPages.Item(0).FolderName = GetFolderPath(folder)
   Marshal.ReleaseComObject(folder)
   Marshal.ReleaseComObject(ns)
End Sub
```

See the code of the GetFolderPath function in FolderPath Property Is Missing in Outlook 2000 and XP.

In order to control the events for the folder, add a checkbox to the page and handle its CheckedChanged event as well as the Dirty, Apply, and Load events of the page as follows:

```
...
Friend WithEvents CheckBox1 As System.Windows.Forms.CheckBox
Private TrackStatusChanges As Boolean

...
```

```
Private Sub CheckBox1_CheckedChanged( _
ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles CheckBox1.CheckedChanged
    If Not TrackStatusChanges Then _
        Me.OnStatusChange() 'this enables the Apply button
End Sub

Private Sub PropertyPage1_Dirty( _
ByVal sender As System.Object, _
ByVal e As AddinExpress.MSO.ADXDirtyEventArgs) Handles MyBase.Dirty
    e.Dirty = True
End Sub

Private Sub PropertyPage1_Apply( _
ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles MyBase.Apply
    CType(AddinModule.CurrentInstance, MyOutlookAddin1.AddinModule) _
        .IsFolderTracked = Me.CheckBox1.Checked
End Sub

Private Sub PropertyPage1_Load( _
ByVal sender As Object, _
ByVal e As System.EventArgs) Handles Me.Load
    TrackStatusChanges = True
    Me.CheckBox1.Checked = _
        CType(AddinModule.CurrentInstance, MyOutlookAddin1.AddinModule) _
            .IsFolderTracked
    TrackStatusChanges = False
End Sub
```

Finally, you add the following property to the add-in module:

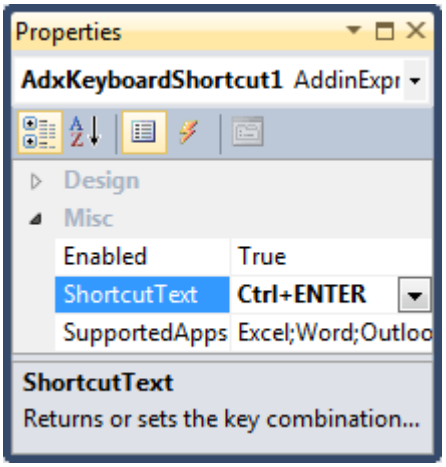
```
Friend Property IsFolderTracked() As Boolean
   Get
      Return ItemsEvents.IsConnected
   End Get
   Set(ByVal value As Boolean)
      If value Then
         ItemsEvents.ConnectTo(ADXOlDefaultFolders.olFolderInbox, True)
      Else
         ItemsEvents.RemoveConnection()
      End If
   End Set
End Property
```

This sample describes adding a property page to the *Folder Properties* dialog for a given folder. To add a property page to the *Tools* | *Options* dialog box (Outlook 2000-2007), you use the PageType and PageTitle properties of the add-in module. In Outlook 2010 that dialog is located at the following UI path: *File Tab* | *Options* | *Add-ins* | *Add-in Options*.

See also Outlook Property Page.

Step #15 – Intercepting Keyboard Shortcuts

To intercept a keyboard shortcut, you use the *Add Keyboard Shortcut* command to add an ADXKeyboardShortcut to the add-in module.



Then, in the *Properties* window for the *Keyboard Shortcut* component, you choose (or enter) the desired shortcut in the ShortcutText property.

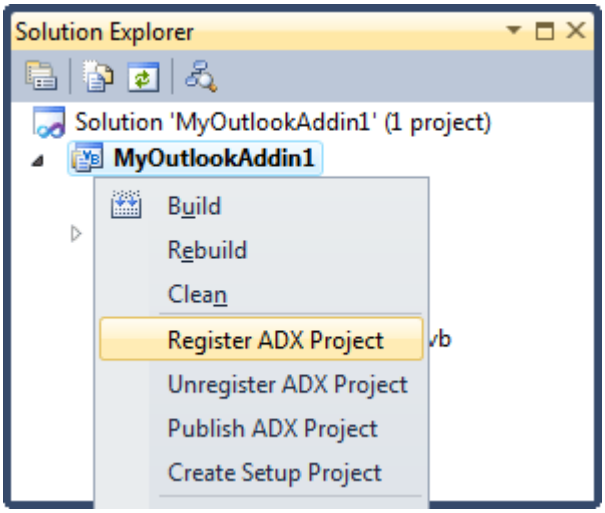
To use keyboard shortcuts, you need to set the HandleShortcuts property of the add-in module to true.

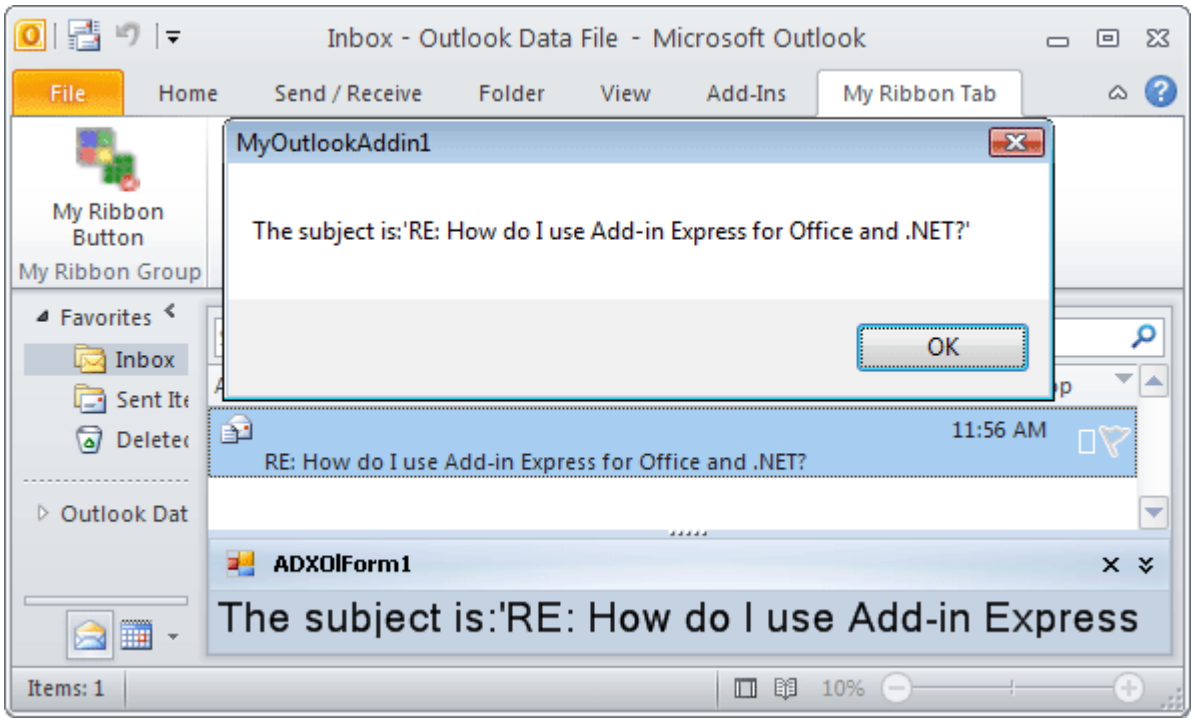
Now you handle the Action event of the component:

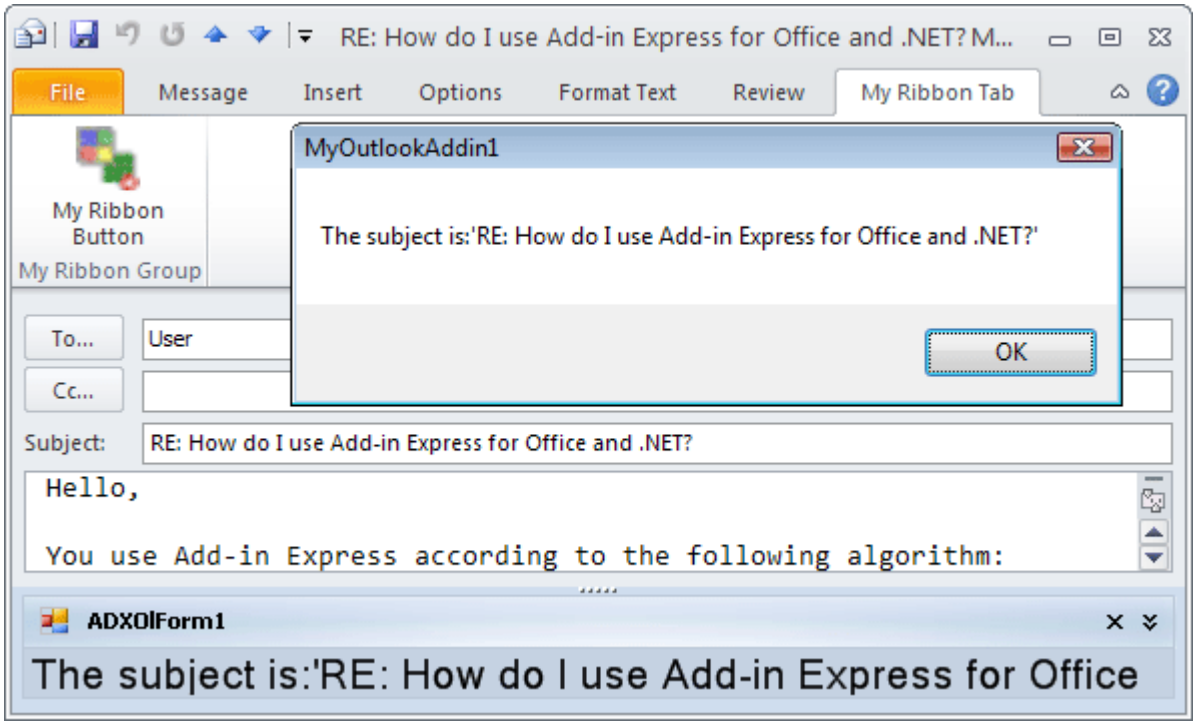
```
Private Sub AdxKeyboardShortcut1_Action(ByVal sender As System.Object) _
   Handles AdxKeyboardShortcut1.Action
   MsgBox("You've pressed " + _
      CType(sender, AddinExpress.MSO.ADXKeyboardShortcut).ShortcutText)
End Sub
```

Step #16 – Running the COM Add-in

Choose *Register Add-in Express Project* in the *Build* menu (if you use the Express edition of Visual Studio, this item can be found in the context menu of the add-in module's designer surface), then restart Outlook and find your option page(s), command bars, and controls, Ribbon controls, and custom task panes.



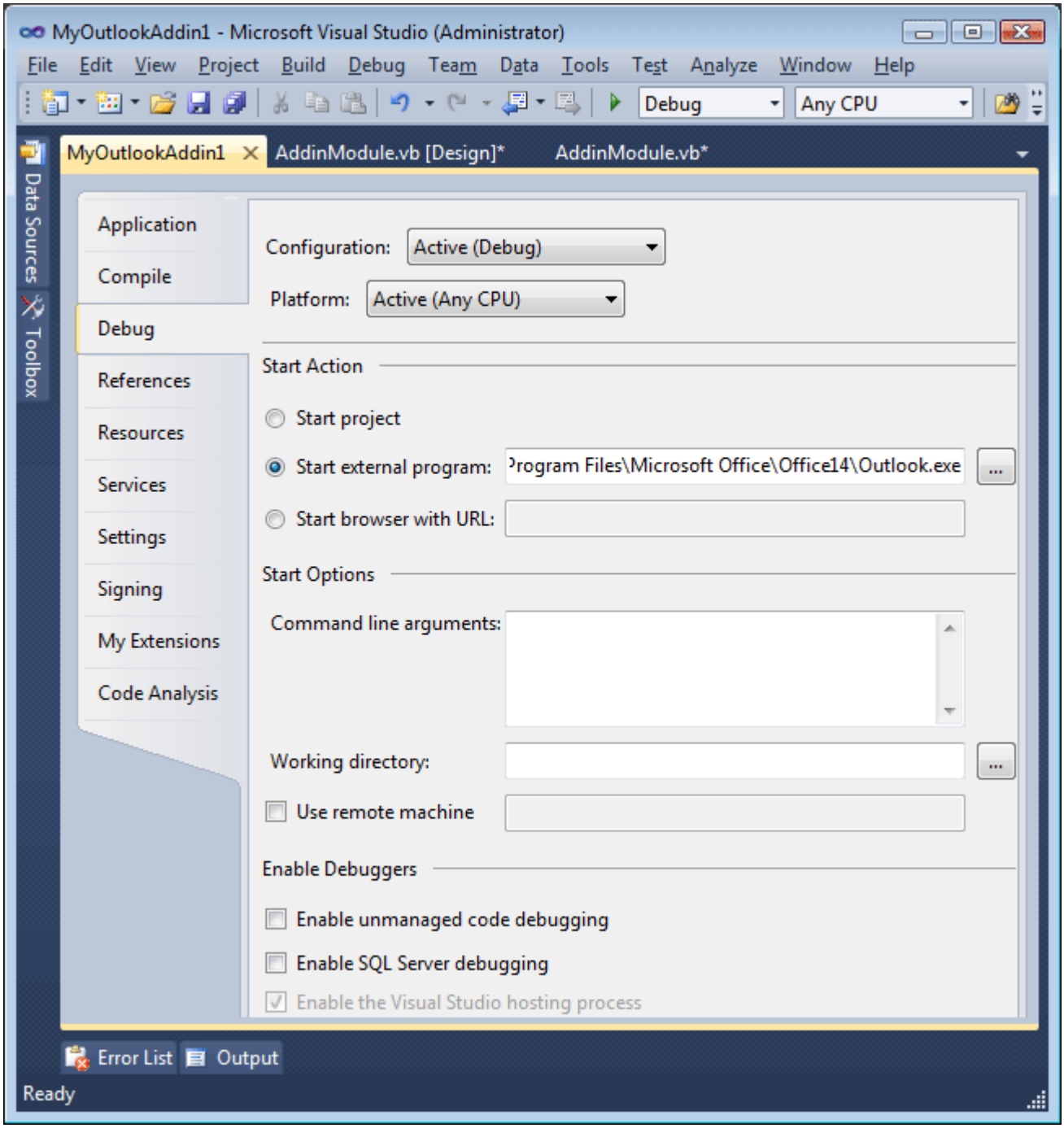


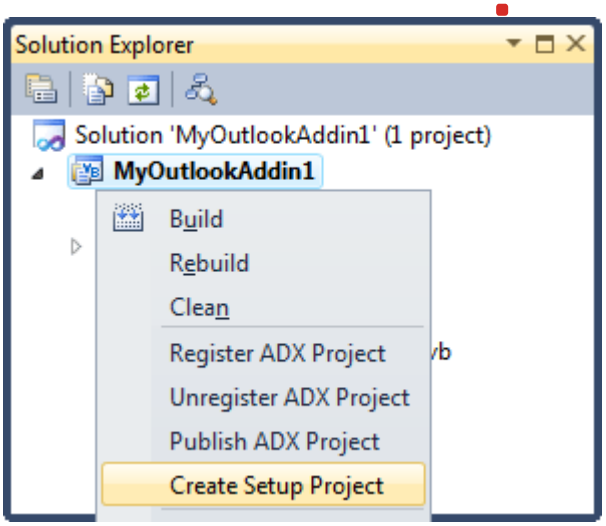


You can find your add-in in the COM Add-ins Dialog.

Step #17 – Debugging the COM Add-in

To debug your add-in, just specify the Outlook executable in *Start External Program* in the *Project Options* window and press *{F5}*.





Step #18 – Deploying the COM Add-in

Create a setup project, build it, copy all setup files to the target PC and run the installer (see Deploying Office Extensions).

The Deploying Add-in Express Projects section describes MSI-based and ClickOnce Deployment. You can also find some useful tips in the Debugging and Deploying section.

Your First .NET Control on an Office Toolbar

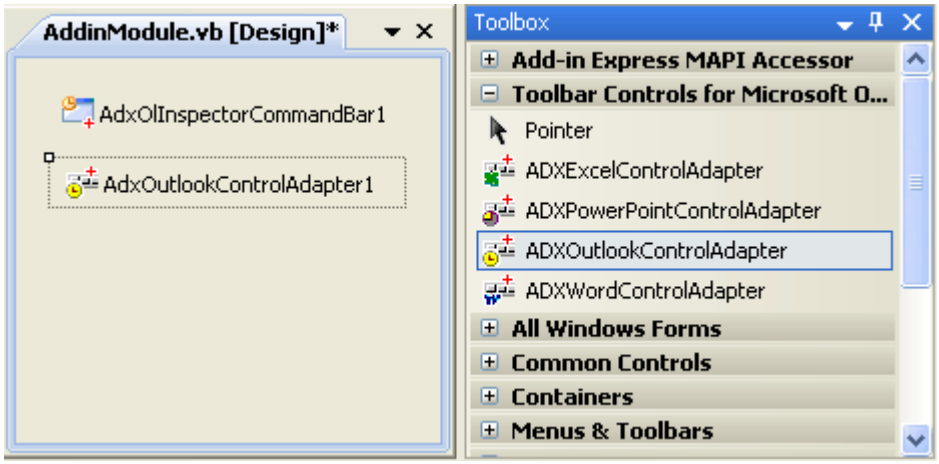
This sample demonstrates features described in Toolbar Controls for Microsoft Office.

Just follow the first three steps described in Your First Microsoft Outlook COM Add-in. Add an ADXOlInspectorCommandBar to the add-in module (see Step #7 – Adding a New Inspector Command Bar of the same sample). Now set adxMsoBarBottom to the Position property of the added command bar.

Step #1 - Adding a Control Adapter

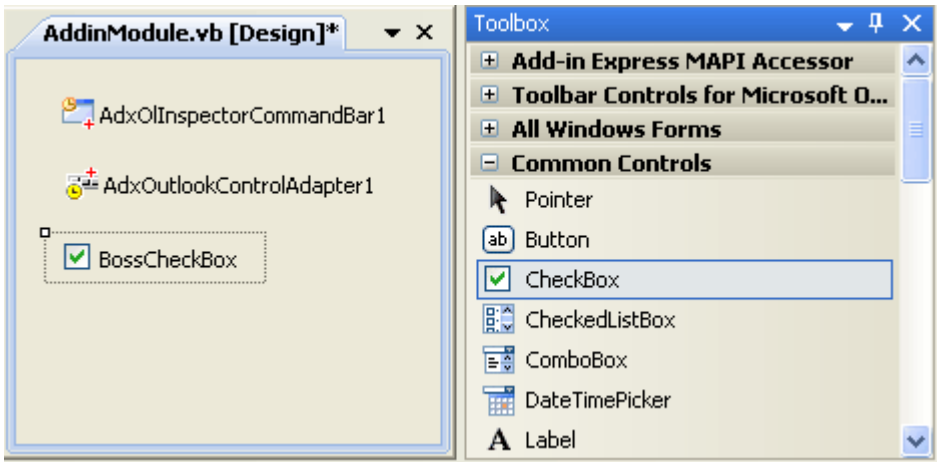
Toolbar Controls for Microsoft Office supports Office applications through special components that we call control adapters. You can find them on the *Toolbar Controls for Microsoft Office* tab in the Toolbox.

The first step in using non-Office controls in your add-in is adding the corresponding control adapter to your add-in module. In this case, we use an ADXOutlookControlAdapter.



Step #2 - Adding Your Control

The add-in module can contain any components including controls. Therefore, you can add a check box (BossCheckBox) directly to your add-in module and customize the checkbox in any way you like.



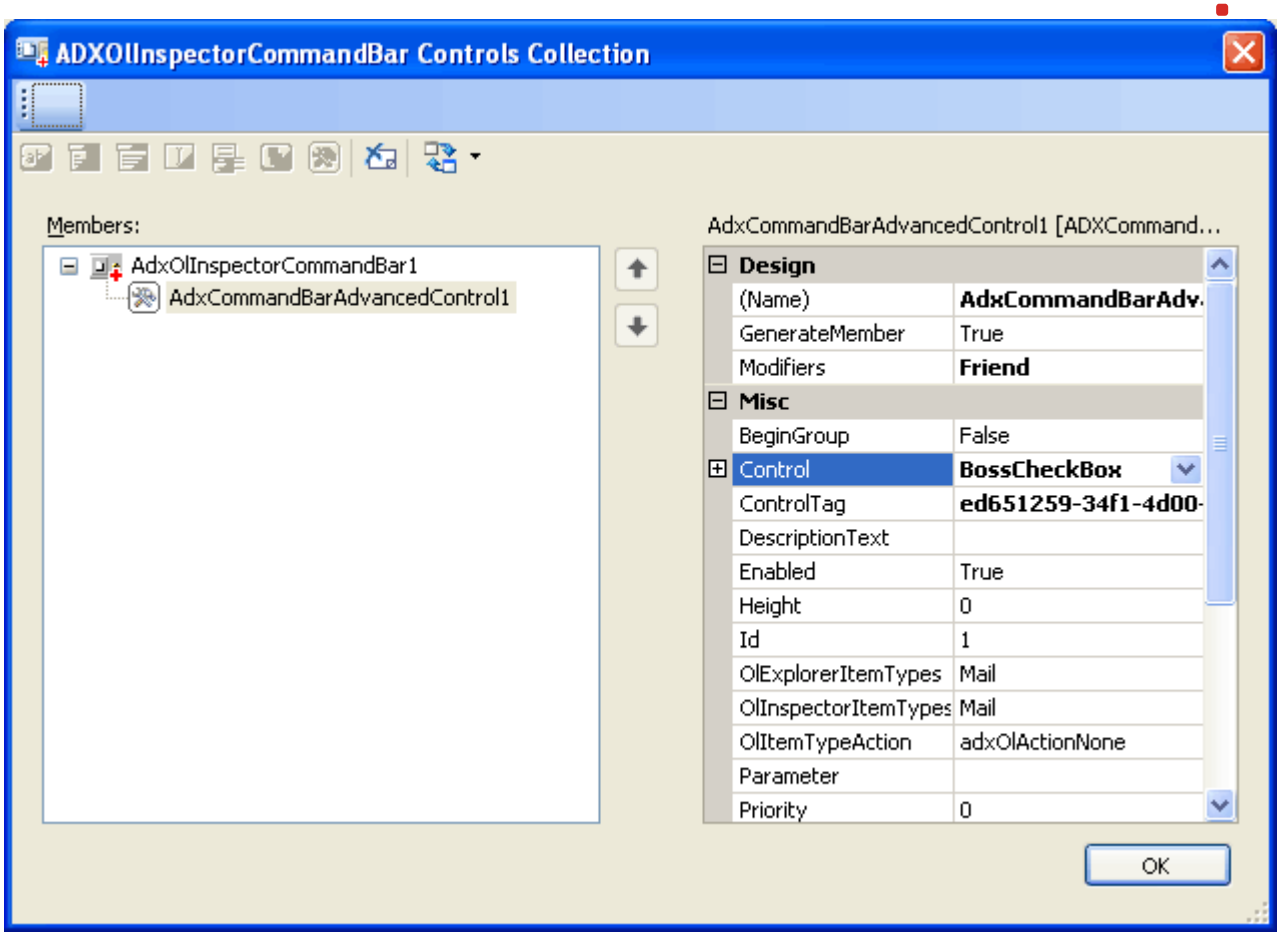
Step #3 - Handling Your Control

To BCC a message to your boss you need to handle the checkbox. You can use the following code to BCC messages. Please note that we do not cover Outlook programming here.

```
Private Sub BossCheckbox_CheckedChanged( _
   ByVal sender As System.Object, ByVal e As System.EventArgs)
   Dim Inspector As Outlook.Inspector = OutlookApp.ActiveInspector
   Dim Item As Outlook.MailItem = _
      CType(Inspector.CurrentItem, Outlook.MailItem)
   Dim currentBossCheckBoxInstance As CheckBox = _
      CType(AdxCommandBarAdvancedControl1.ActiveInstance, CheckBox)
   If currentBossCheckBoxInstance.Checked Then
      Item.BCC = "myboss@mydomain.com"
   Else
      Item.BCC = ""
   End If
   Marshal.ReleaseComObject(Item)
   Marshal.ReleaseComObject(Inspector)
End Sub
```

Step #4 - Binding Your Control to the CommandBar

To bind BossCheckBox to the command bar, you add an advanced command bar control (ADXCommandBarAdvancedControl1) to the Controls collection of your command bar and select BossCheckBox in the Control property of the ADXCommandBarAdvancedControl1. That's all.



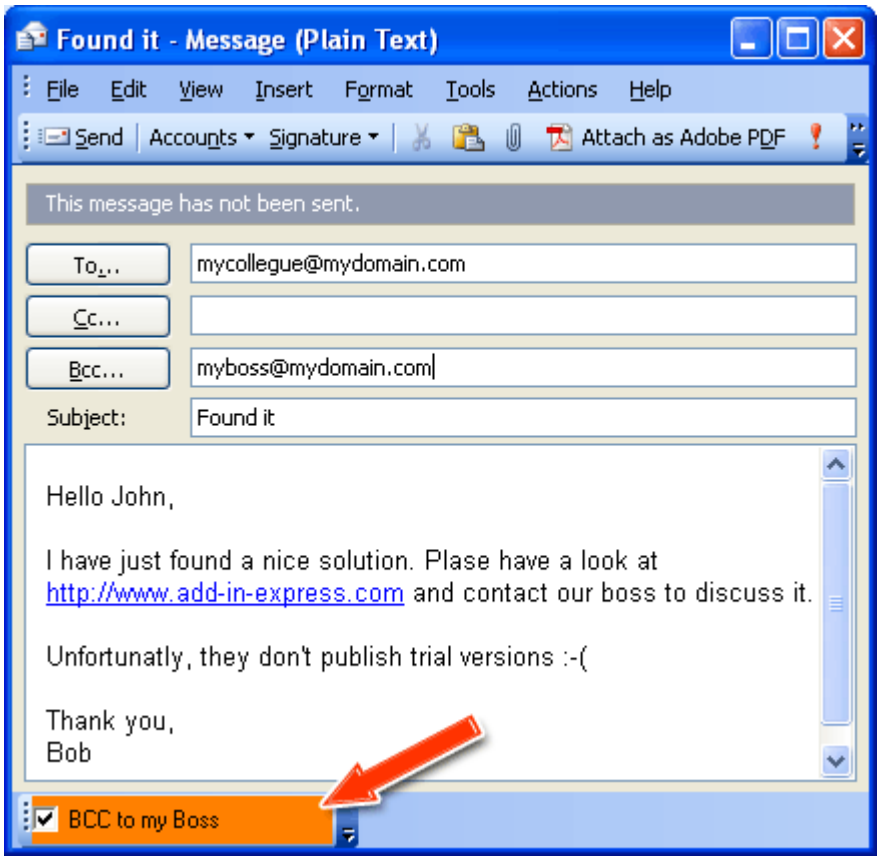
Below we give the complete InitializeComponent method of our add-in module that relates to our example:

```
  Private Sub InitializeComponent()
    Me.components = New System.ComponentModel.Container
    Me.AdxAddinAdditionalModuleItem1 = New _
      AddinExpress.MSO.ADXAddinAdditionalModuleItem(Me.components)
    Me.AdxOlInspectorCommandBar1 = New _
      AddinExpress.MSO.ADXOlInspectorCommandBar(Me.components)
    Me.AdxOutlookControlAdapter1 = New _
      AddinExpress.ToolbarControls.ADXOutlookControlAdapter(Me.components)
    Me.BossCheckBox = New System.Windows.Forms.CheckBox
    Me.AdxCommandBarAdvancedControl1 = New _
      AddinExpress.MSO.ADXCommandBarAdvancedControl(Me.components)
    '
    'AdxOlInspectorCommandBar1
    '
    Me.AdxOlInspectorCommandBar1.CommandBarName = _
      "AdxOlInspectorCommandBar1"
    Me.AdxOlInspectorCommandBar1.CommandBarTag = _
      "77fc20e0-bf9e-47d0-997f-eb1167f506a4"
    Me.AdxOlInspectorCommandBar1.Controls.Add _
```

```
        (Me.AdxCommandBarAdvancedControl1)
    Me.AdxOlInspectorCommandBar1.Position = _
       AddinExpress.MSO.ADXMsoBarPosition.adxMsoBarBottom
    Me.AdxOlInspectorCommandBar1.Temporary = True
    Me.AdxOlInspectorCommandBar1.UpdateCounter = 4
    '
    'BossCheckBox
    '
    Me.BossCheckBox.BackColor = _
       System.Drawing.Color.FromArgb(CType(CType(255, Byte), Integer), _
       CType(CType(128, Byte), Integer), CType(CType(0, Byte), Integer))
    Me.BossCheckBox.AutoSize = True
    Me.BossCheckBox.Location = New System.Drawing.Point(0, 0)
    Me.BossCheckBox.Name = "BossCheckBox"
    Me.BossCheckBox.Size = New System.Drawing.Size(104, 24)
    Me.BossCheckBox.TabIndex = 0
    Me.BossCheckBox.Text = "BCC to my Boss"
    Me.BossCheckBox.UseVisualStyleBackColor = True
    '
    'AdxCommandBarAdvancedControl1
    '
    Me.AdxCommandBarAdvancedControl1.Control = Me.BossCheckBox
    Me.AdxCommandBarAdvancedControl1.ControlTag = _
       "ed651259-34f1-4d00-8716-e56ccf0118d4"
    Me.AdxCommandBarAdvancedControl1.Temporary = True
    Me.AdxCommandBarAdvancedControl1.UpdateCounter = 3
    '
    'AddinModule
    '
    Me.AddinName = "MyAddin"
    Me.SupportedApps = AddinExpress.MSO.ADXOfficeHostApp.ohaOutlook

End Sub
```

Step #5 – Register and Run Your Add-in

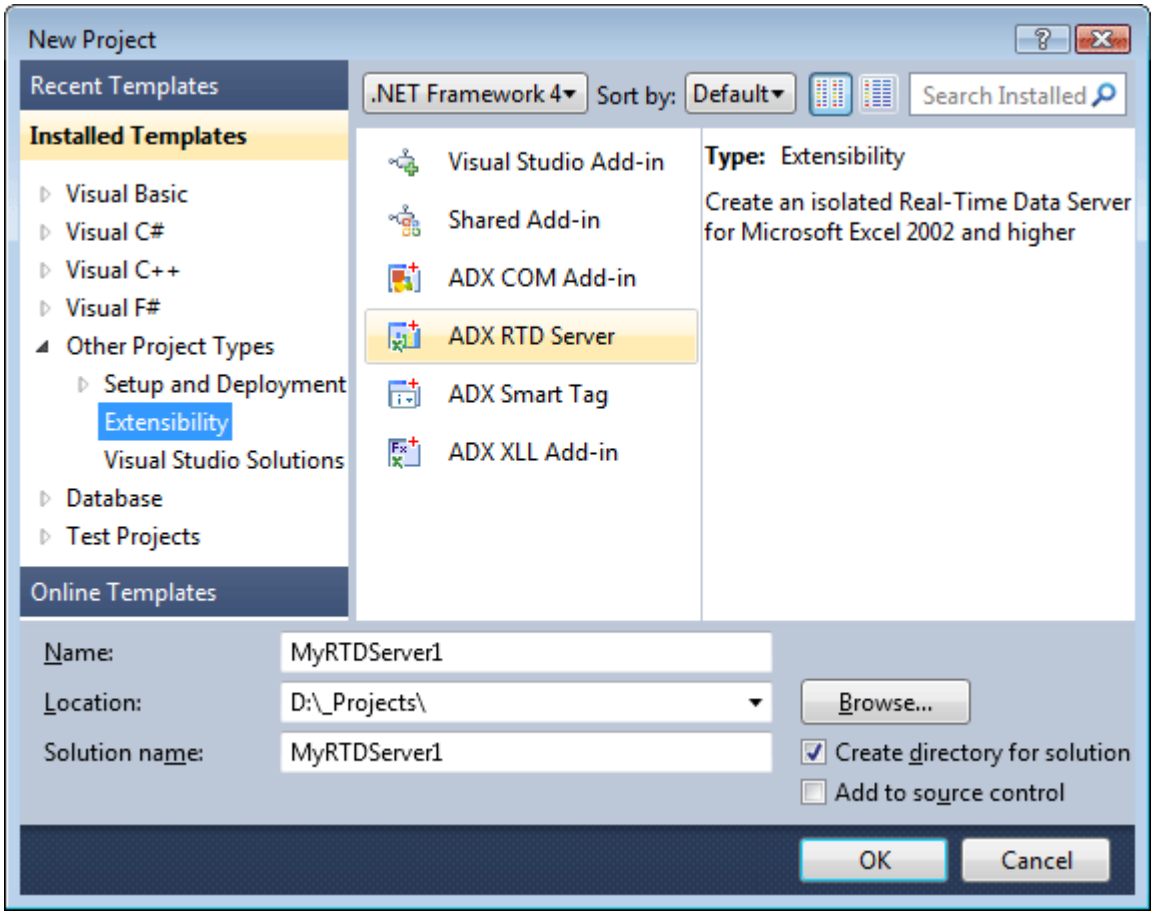
Finally, you can rebuild the add-in project, run Outlook, and find your check box:


Found it - Message (Plain Text)
File Edit View Insert Format Tools Actions Help
Send Accounts Signature Attach as Adobe PDF
This message has not been sent.
To...
mycollegue@mydomain.com
Cc...
Bcc...
myboss@mydomain.com
Subject:
Found it
Hello John,
I have just found a nice solution. Plase have a look at
http://www.add-in-express.com and contact our boss to discuss it.
Unfortunatly, they don't publish trial versions :-(
Thank you,
Bob
BCC to my Boss

Your First Excel RTD Server

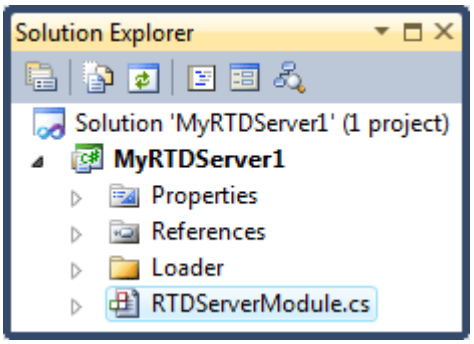
Step #1 – Creating a New RTD Server Project

Choose the *Add-in Express RTD Server* project template in the New Project dialog.

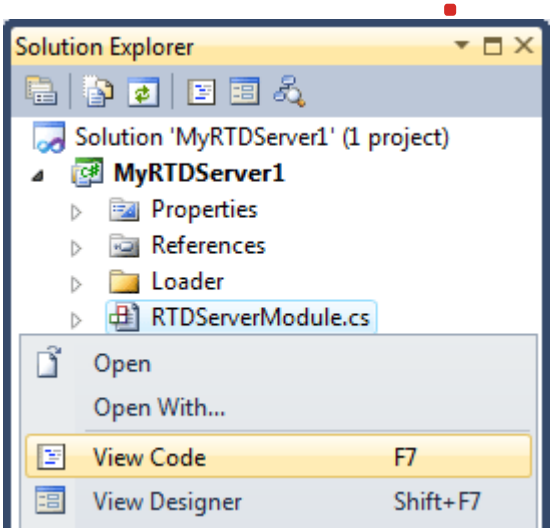


When you click OK, the RTD server project wizard starts. In the wizard window, you choose the programming language and specify a strong name key file to use.

The project wizard creates and opens a new solution in the IDE. The solution includes the RTD server project containing the RTDServerModule.vb (or RTDServerModule.cs) file discussed in the next step.



Step #2 - RTD Server Module



RTDServerModule.vb (or RTDServerModule.cs) is the core part of the RTD server project. The module is a container for ADXRTDTopic components. It is a descendant of the ADXRTDServerModule class implementing the IRtdServer COM interface and allowing you to manage server's topics and their code. To review its source code, right-click the file in the *Solution Explorer* and choose *View Code* in the context menu.

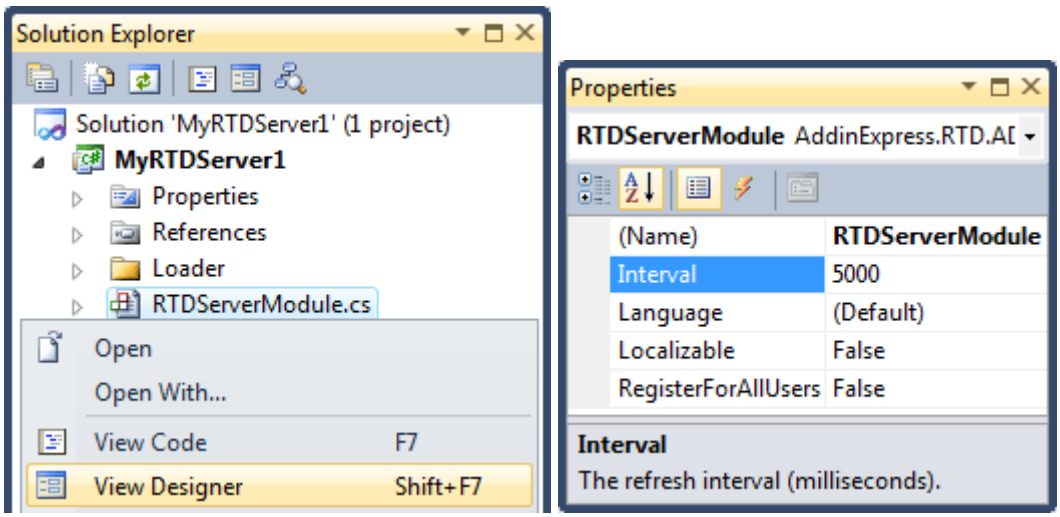
The code of RTDServerModule.vb is as follows:

```
Imports System.Runtime.InteropServices
Imports System.ComponentModel

'Add-in Express RTD Server Module
<GuidAttribute("ACD23E21-2F2B-4C13-B894-6C74D8AD2EEE"), _
   ProgIdAttribute("MyRTDServer1.RTDServerModule")> _
Public Class RTDServerModule
   Inherits AddinExpress.RTD.ADXRTDServerModule

#Region " Component Designer generated code. "
   'Required by designer
   Private components As System.ComponentModel.IContainer

   'Required by designer - do not modify
   'the following method
   Private Sub InitializeComponent()
      Me.components = New System.ComponentModel.Container
   End Sub

#End Region

#Region " Add-in Express automatic code "

   'Required by Add-in Express - do not modify
   'the methods within this region

   Public Overrides Function GetContainer() _
      As System.ComponentModel.IContainer
      If components Is Nothing Then
         components = New System.ComponentModel.Container
      End If
      GetContainer = components
   End Function
```

```
   <ComRegisterFunctionAttribute()> _
   Public Shared Sub RTDServerRegister(ByVal t As Type)
      AddinExpress.RTD.ADXRTDServerModule.ADXRTDServerRegister(t)
   End Sub

   <ComUnregisterFunctionAttribute()> _
   Public Shared Sub RTDServerUnregister(ByVal t As Type)
      AddinExpress.RTD.ADXRTDServerModule.ADXRTDServerUnregister(t)
   End Sub

#End Region

   Public Sub New()
      MyBase.New()
      'This call is required by the Component Designer
      InitializeComponent()

      'Add any initialization after the InitializeComponent() call
   End Sub
End Class
```

Step #3 – Add-in Express RTD Server Designer

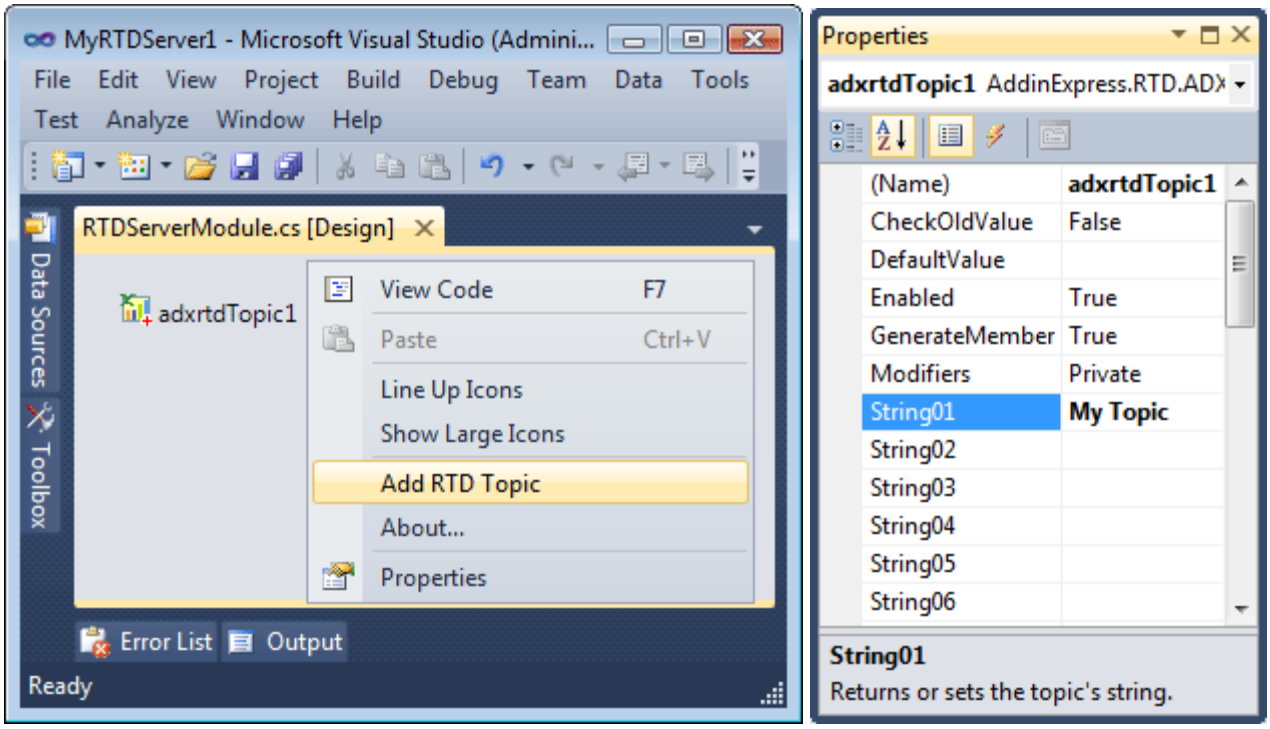
The module designer allows setting RTD server properties and adding components to the module. In the *Solution Explorer*, right-click the RTDServerModule.vb (or RTDServerModule.cs) file and choose the *View Designer* popup menu item.



In the Properties window, you set properties of your RTD server.

Step #4 - Adding and Handling a New Topic

To add a new topic to your RTD server, you use the *Add RTD Topic* command that adds a new ADXRTDTopic component to the module (see RTD Topic). Select the newly added component and, in the Properties window, specify the topic using the String## properties.

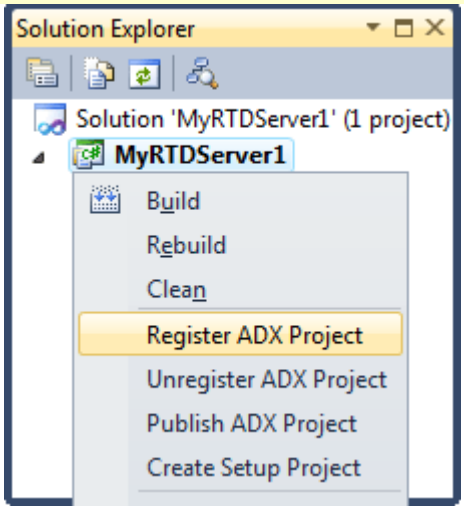


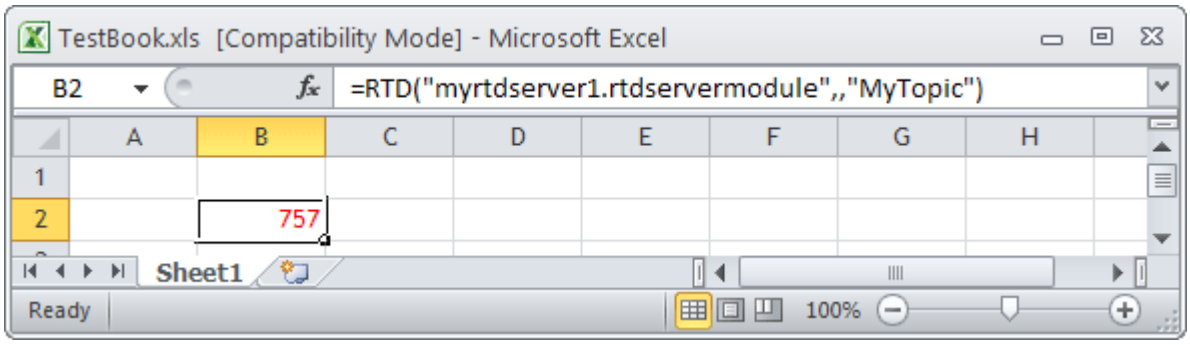
To handle the RefreshData event of the RTD Topic component, add the RefreshData event handler and write your code to handle the event:

```
Private Function AdxrtdTopic1_RefreshData( _
   ByVal sender As System.Object) As System.Object _
      Handles AdxrtdTopic1.RefreshData
   Dim Rnd As New System.Random
   Return Rnd.Next(2000)
End Function
```

Step #5 - Running the RTD Server

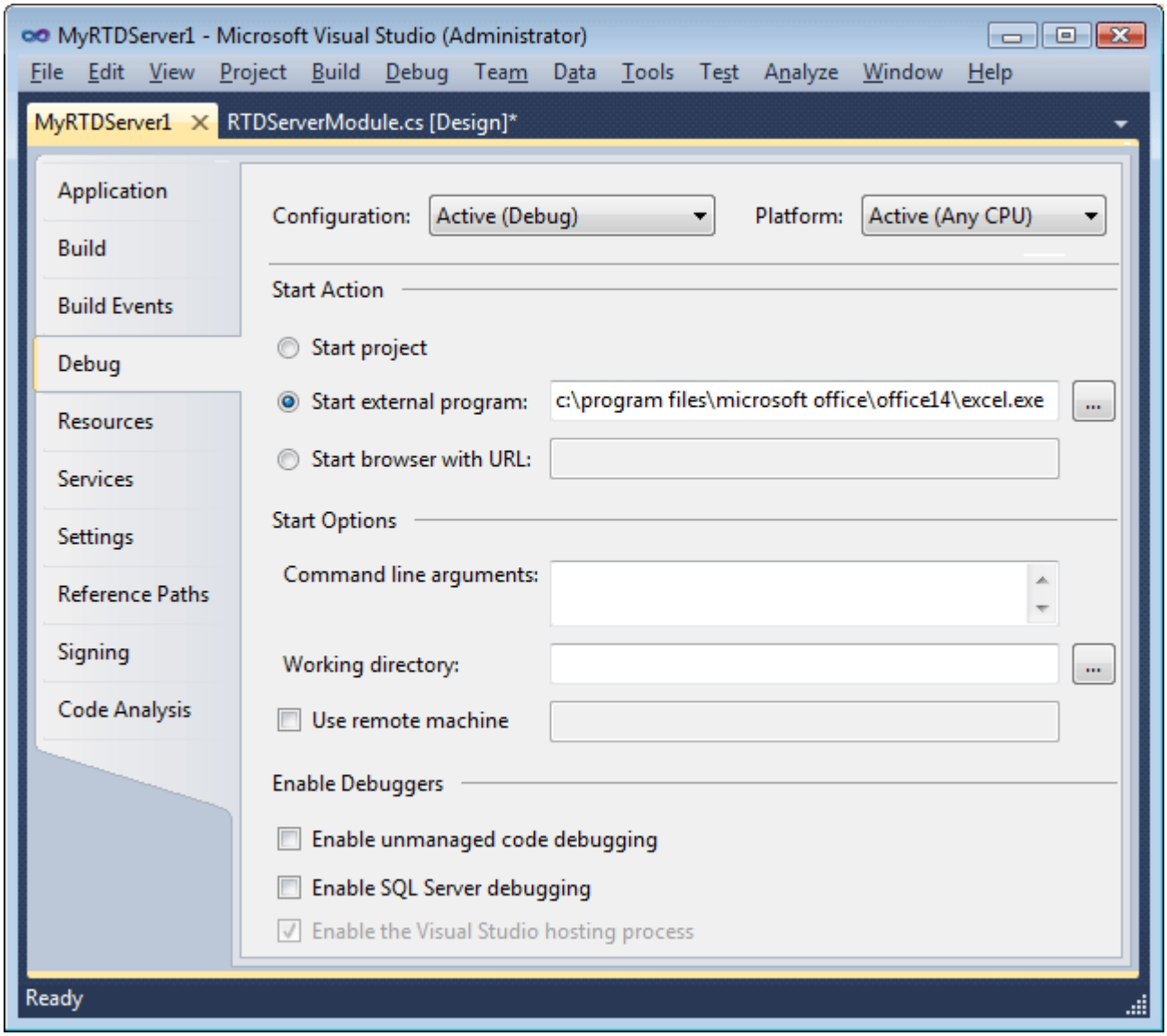
Choose the *Register Add-in Express Project* item in the Build menu (if you use the Express edition of Visual Studio, this item can be found in the context menu of the add-in module's designer surface), restart Excel, and enter the RTD function to a cell (see below). See *Control Panel* | *Regional Settings* for the parameters separator.





Step #6 - Debugging the RTD Server

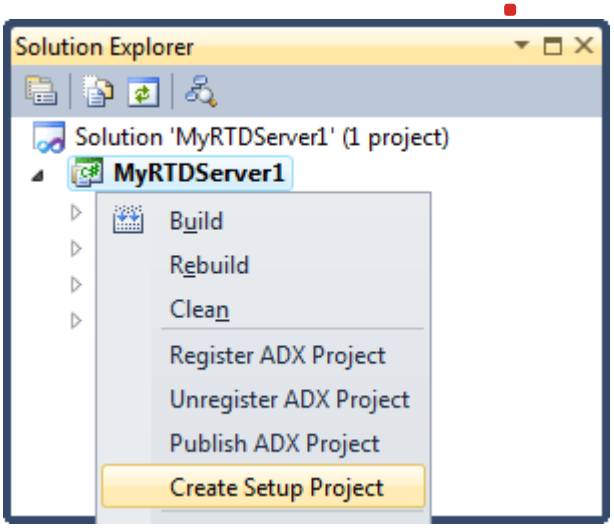
To debug your RTD server, just specify Excel as the *Start Program* in the *Project Options* window.



Step #7 - Deploying the RTD Server

Create a setup project, build it, copy all setup files to the target PC and run the installer (see Deploying Office Extensions).

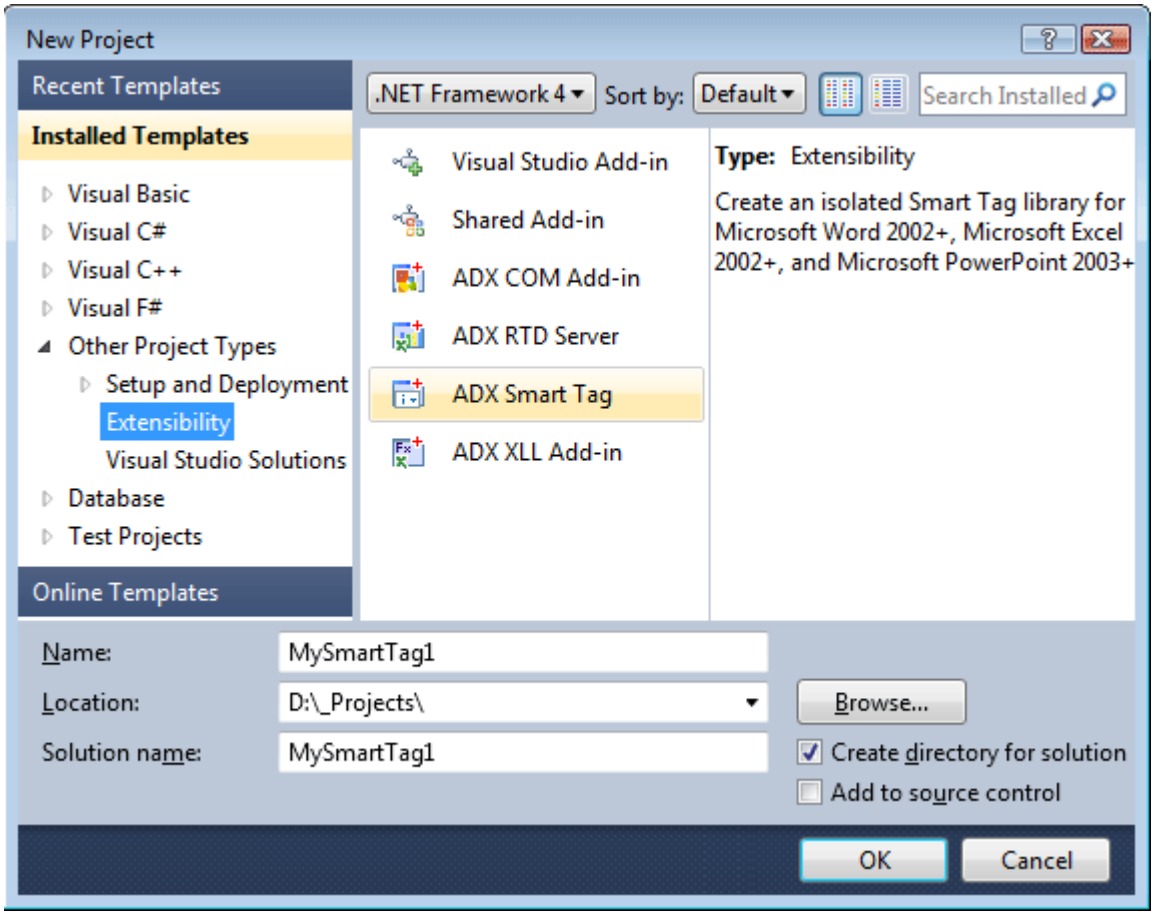
See also Deploying Add-in Express Projects and RTD.



Your First Smart Tag

Step #1 – Creating a New Smart Tag Library Project

Choose the *Add-in Express Smart Tag* project template in the New Project dialog.



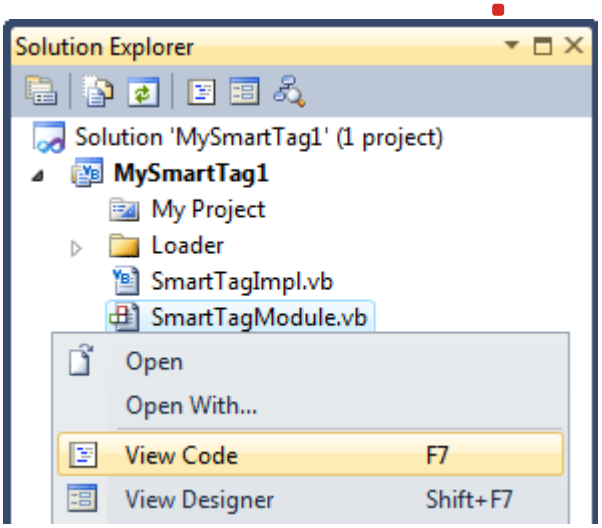
Click OK to start the Smart Tag project wizard. In the wizard window, you choose the programming language and specify a strong name key file to use.

The project wizard creates and opens a new solution in IDE. The solution contains the smart tag project.

> Do not delete the SmartTagImpl.vb (SmartTagImpl.cs) file required by the Add-in Express implementation of the Smart Tag technology. Usually, you do not need to modify it.

The smart tag project contains the SmartTagModule.vb (or SmartTagModule.cs) file discussed in the next step.

Step #2 - Smart Tag Module



SmartTagModule.vb (or SmartTagModule.cs) is a smart tag module that is the core part of the smart tag project. The module is a container for ADXSmartTag components. It contains the SmartTagModule class, a descendant of ADXSmartTagModule, which implements the interfaces required by the Smart Tag technology and allows managing smart tags. To review its source code, right-click the file in *Solution Explorer* and choose *View Code* in the popup menu.

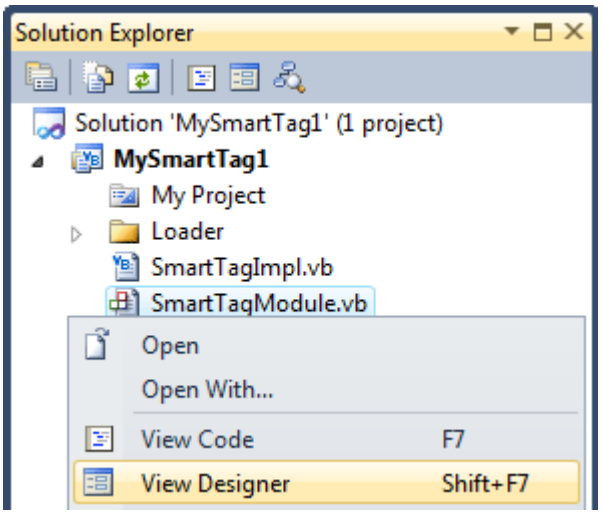
The smart tag module contains the following code:

```
Imports System.Runtime.InteropServices
Imports System.ComponentModel

'Add-in Express Smart Tag Module
<GuidAttribute("6FFEFD7D-FD3C-47EE-AE67-A84DE4AD4E7A"),
       ProgIdAttribute("MySmartTag1.SmartTagModule")> _
Public Class SmartTagModule
    Inherits AddinExpress.SmartTag.ADXSmartTagModule

#Region " Component Designer generated code. "
   'Required by designer
   Private components As System.ComponentModel.IContainer

   'Required by designer - do not modify
   'the following method
   Private Sub InitializeComponent()
      Me.components = New System.ComponentModel.Container
      '
      'SmartTagModule
      '
      Me.NamespaceURI = "mysmarttag1/smarttagmodule"
      Me.LibraryName = New AddinExpress.SmartTag.ADXLocalizedList
      Me.LibraryName.Add(0, "MySmartTag1")
      Me.Description = New AddinExpress.SmartTag.ADXLocalizedList
      Me.Description.Add(0, "This is a description")

   End Sub
#End Region

#Region " Add-in Express automatic code "

   'Required by Add-in Express - do not modify
```

```
   'the methods within this region
   Public Overrides Function GetContainer() As
      System.ComponentModel.IContainer
      If components Is Nothing Then
         components = New System.ComponentModel.Container
      End If
      GetContainer = components
   End Function

   <ComRegisterFunctionAttribute()> _
   Public Shared Sub SmartTagRegister(ByVal t As Type)
      AddinExpress.SmartTag.ADXSmartTagModule.ADXSmartTagRegister(t, _
            System.Type.GetType("MySmartTag1.SmartTagRecognizerImpl"), _
            System.Type.GetType("MySmartTag1.SmartTagActionImpl"))
   End Sub

   <ComUnregisterFunctionAttribute()> _
   Public Shared Sub SmartTagUnregister(ByVal t As Type)
      AddinExpress.SmartTag.ADXSmartTagModule.ADXSmartTagUnregister(t, _
            System.Type.GetType("MySmartTag1.SmartTagRecognizerImpl"), _
            System.Type.GetType("MySmartTag1.SmartTagActionImpl"))
   End Sub
#End Region

   Public Sub New()
      MyBase.New()

      'This call is required by the Component Designer
      InitializeComponent()
      'Add any initialization after the InitializeComponent() call
   End Sub
End Class
```

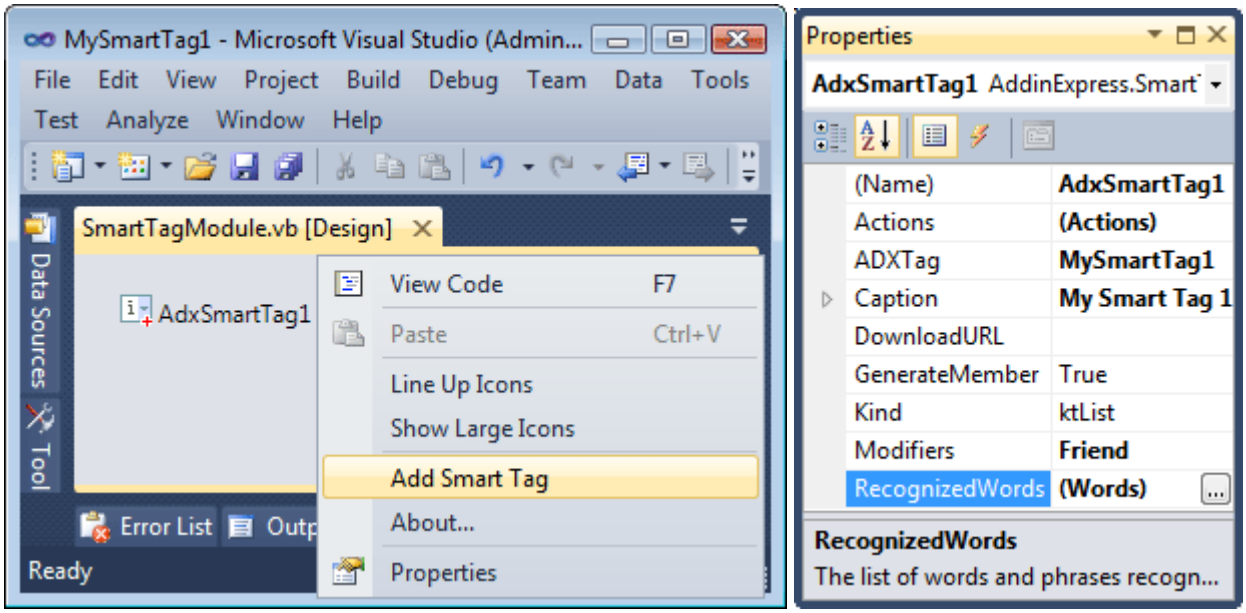
Step #3 - Smart Tag Designer

The module designer allows setting smart tag properties and adding components to the module. In *Solution Explorer*, right-click the SmartTagModule.vb (or SmartTagModule.cs) file and choose the *View Designer* popup menu item. In the *Properties* window, you set properties of your smart tag module (see Smart Tags).

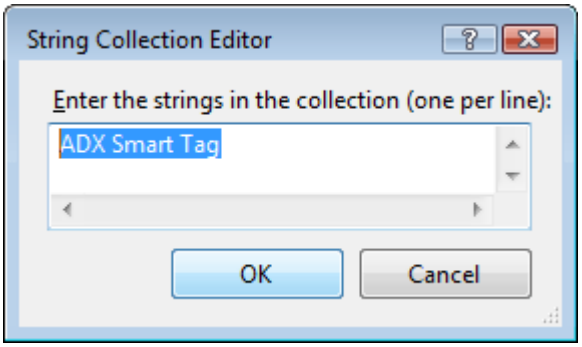


Step #4 – Adding a New Smart Tag

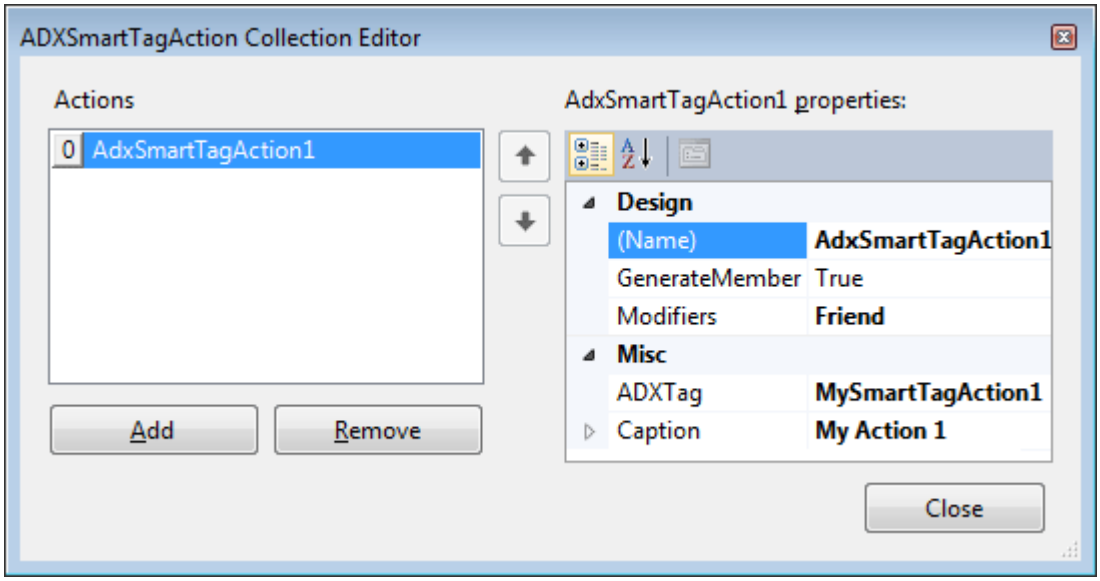
Add a new ADXSmartTag component to the module (see Smart Tags).



Select the newly added component and, in the *Properties* window, specify the caption for the added smart tag in the Caption property. The value of this property will become a caption of the smart tag context menu. Also, specify the phrase(s) recognizable by the smart tag in the RecognizedWords string collection.



Step #5 – Adding and Handling Smart Tag Actions

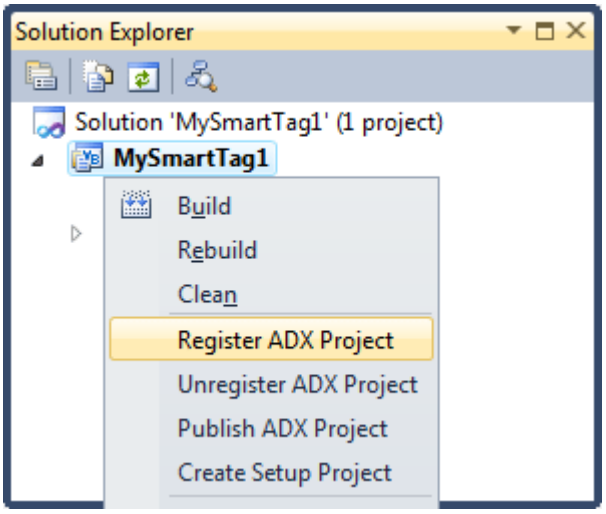


Now you add smart tag actions to your smart tag context menu. To add a new smart tag action, add an item to the Actions collection and set its Caption property that will become the caption of the appropriate item in the smart tag context menu.

To handle the Click event of the action, close the Actions collection editor, and, in the *Properties* window, select the newly added action. Then add the Click event handler and write your code to handle the event:

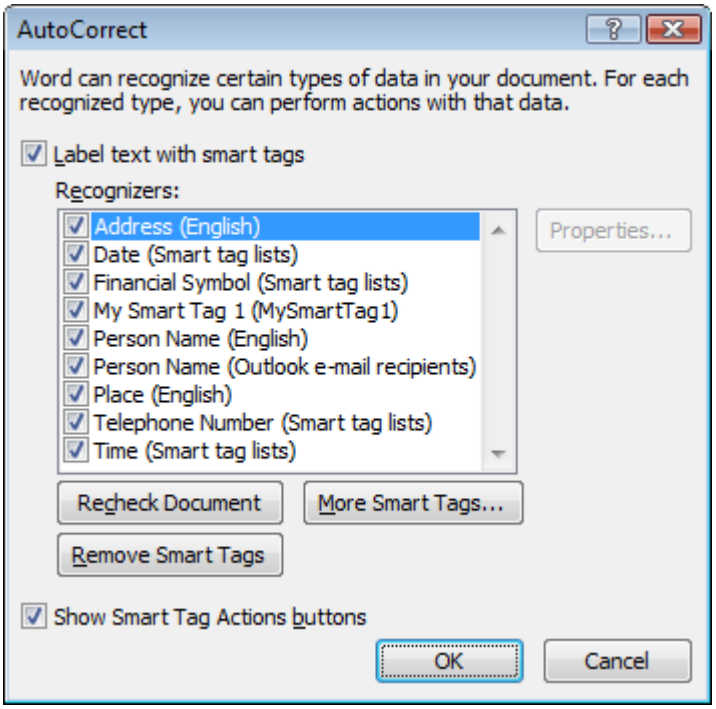
```
Private Sub AdxSmartTagAction1_Click(ByVal sender As System.Object, _
   ByVal e As AddinExpress.SmartTag.ADXSmartTagActionEventArgs) _
      Handles AdxSmartTagAction1.Click
   MsgBox("Recognized text is '" + e.Text + "'!")
End Sub
```

Step #6 - Running Your Smart Tag



Choose the *Register Add-in Express Project* item in the *Build* menu (if you use the Express edition of Visual Studio, this item can be found in the context menu of the add-in module's designer surface). Restart Word, put the words recognizable by your smart tag into a document, and see if the smart tag works.

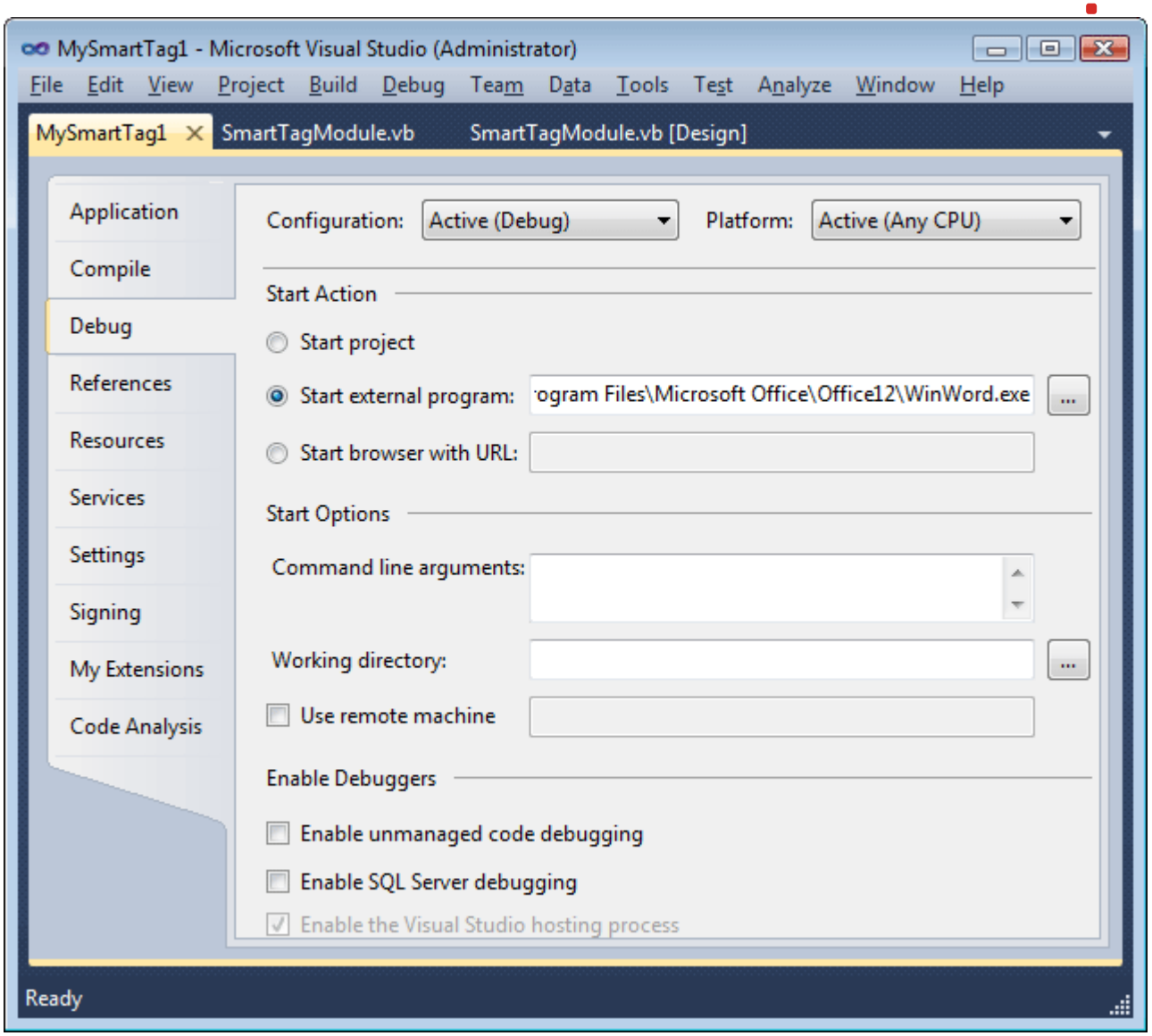
Smart tags are deprecated in Excel 2010 and Word 2010. Though, you can still use the related APIs in projects for Excel 2010 and Word 2010, see Changes in Word 2010 and Changes in Excel 2010.



Also, check if the smart tag is present in the *AutoCorrect* dialog. In Office 2000-2003, choose *Tools* | *AutoCorrect* in the main menu and find your smart tag on the *Smart Tags* tab. In Office 2007, the path to this dialog is as follows: *Office button* | *Word Options* | *Add-ins* | *"Manage" Smart Tags* | *Go*. In Office 2010, see *File tab* | *Options* | *Add-ins* | *"Manage" Actions* | *Go*.

Step #7 - Debugging the Smart Tag

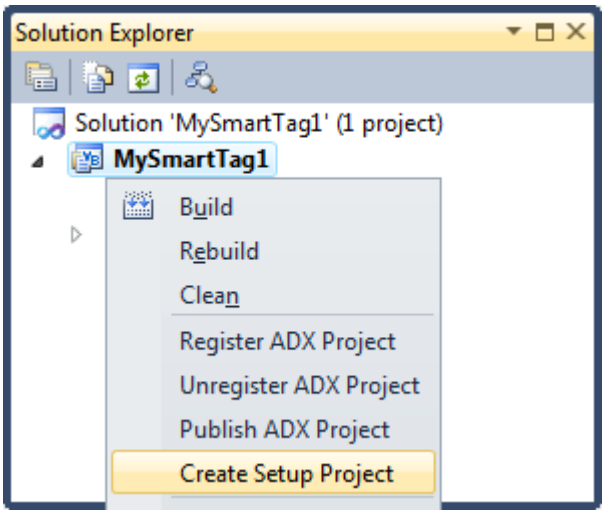
To debug your Smart Tag, just specify the add-in host application as the Start Program in the Project Options window.



Step #8 – Deploying the Smart Tag

Create a setup project, build it, copy all setup files to the target PC and run the installer (see Deploying Office Extensions).

See also Deploying Add-in Express Projects.

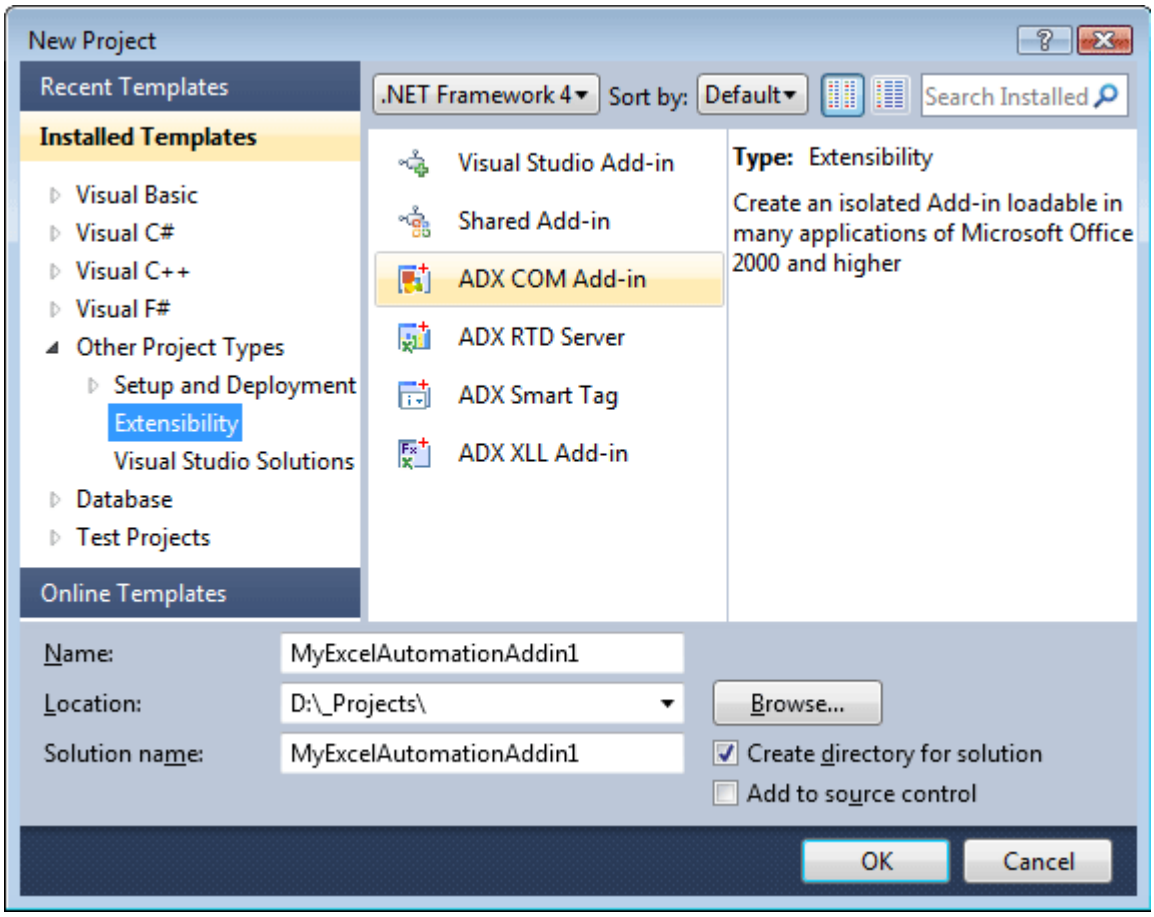


Your First Excel Automation Add-in

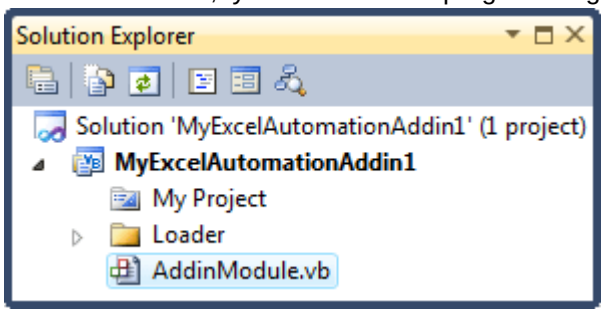
The fact is that Excel Automation Add-ins do not differ from COM Add-ins except for the registration in the registry. That is why Add-n Express bases Excel Automation Add-in projects on COM add-in projects.

Step #1 – Creating a New COM Add-in Project

Choose the *Add-in Express COM Add-in* project template in the New Project dialog.



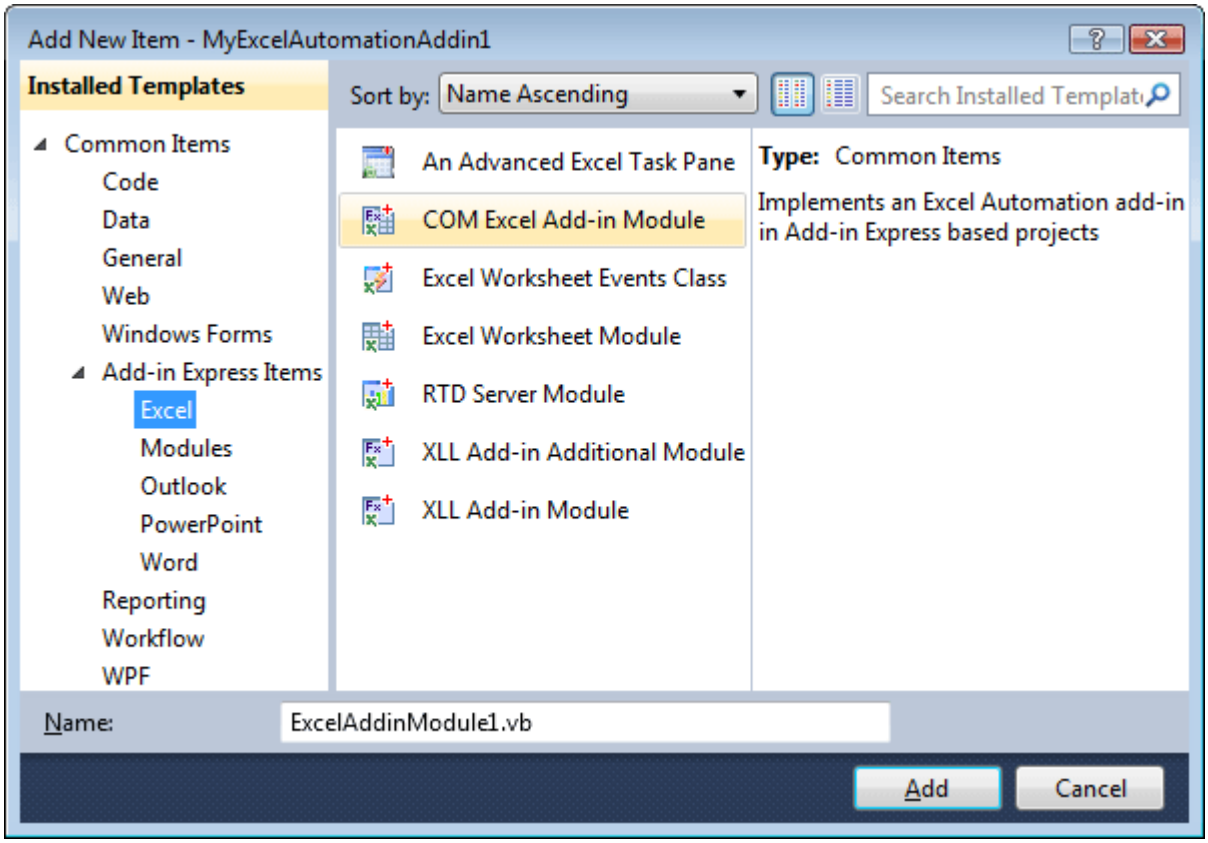
Click OK to start the COM add-in project wizard. In the wizard windows, you choose the programming language, applications supported by your add-in, and interop assemblies to use. Note that since Excel Automation add-ins are supported in Excel 2002 and higher, the minimal Excel interop assembly version is XP (2002).



The wizard creates and opens a new COM Add-in solution in IDE. The solution contains an only project, the COM add-in project. Please refer to Your First Microsoft Office COM Add-in

Step #2 – Adding a New COM Excel Add-in Module

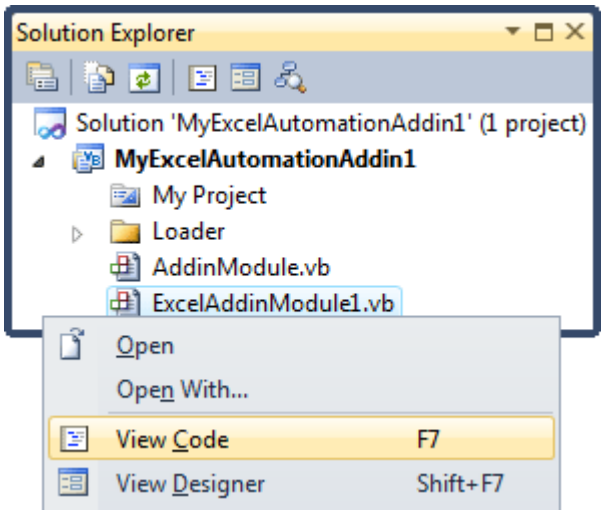
In order to add Excel user-defined functions to the COM add-in, you choose the *COM Excel Add-in Module* in the Add New Item dialog.



This adds the ExcelAddinModule1.vb (or ExcelAddinModule1.cs) file to your COM Add-in project.

Step #3- Writing a User-Defined Function

In Solution Explorer, right-click the ExcelAddinModule.vb (or ExcelAddinModule.cs) file and choose *View Code* in the context menu.



The module contains the following code:

```
Imports System.Runtime.InteropServices

'Add-in Express Excel Add-in Module
<GuidAttribute("287D044F-D233-47E6-BB48-35999635BAD3"), _
```

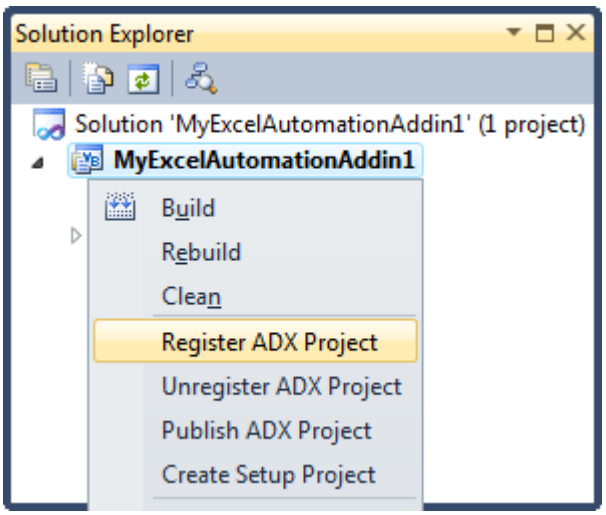
```
ProgIdAttribute("MyExcelAutomationAddin2.ExcelAddinModule1"), _
      ClassInterface(ClassInterfaceType.AutoDual)> _
Public Class ExcelAddinModule1
   Inherits AddinExpress.MSO.ADXExcelAddinModule

#Region " Add-in Express automatic code "

   <ComRegisterFunctionAttribute()> _
   Public Shared Sub AddinRegister(ByVal t As Type)
      AddinExpress.MSO.ADXExcelAddinModule.ADXExcelAddinRegister(t)
   End Sub

   <ComUnregisterFunctionAttribute()> _
   Public Shared Sub AddinUnregister(ByVal t As Type)
      AddinExpress.MSO.ADXExcelAddinModule.ADXExcelAddinUnregister(t)
   End Sub

#End Region

   Public Sub New()
      MyBase.New()
   End Sub
End Class
```

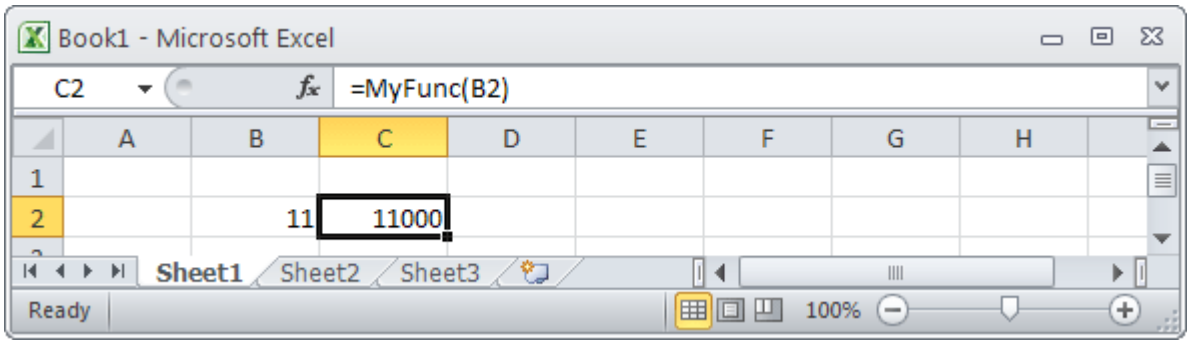
Just add a new public function to the class. Say, the following one:

```
Imports Excel = Microsoft.Office.Interop.Excel
...
Public Function MyFunc(ByVal Range As Object) As Object
   MyFunc = CType(Range, Excel.Range).Value * 1000
End Function
```

Step #4 - Running the Add-in

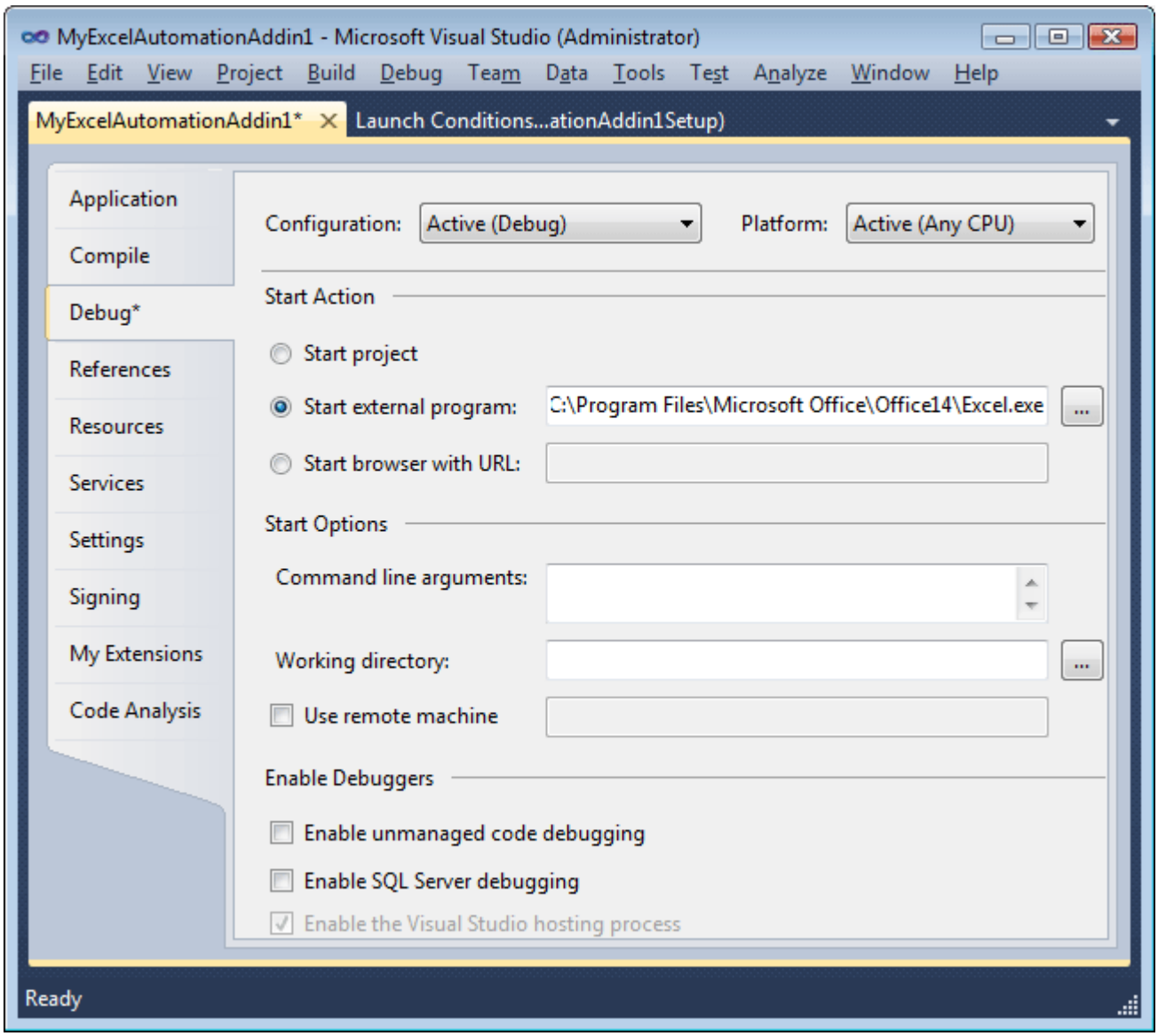
Choose *Register Add-in Express Project* in the Build menu (if you use the Express edition of Visual Studio, this item can be found in the context menu of the add-in module's designer surface), restart Excel, and check if your add-in works.





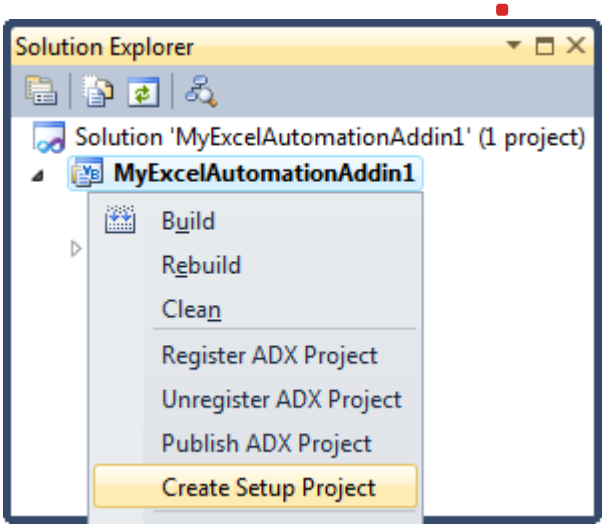
Step #5 - Debugging the Excel Automation Add-in

To debug your add-in, specify Excel as the Start Program in the Project Options window and run the project.



Step #6 - Deploying the Add-in

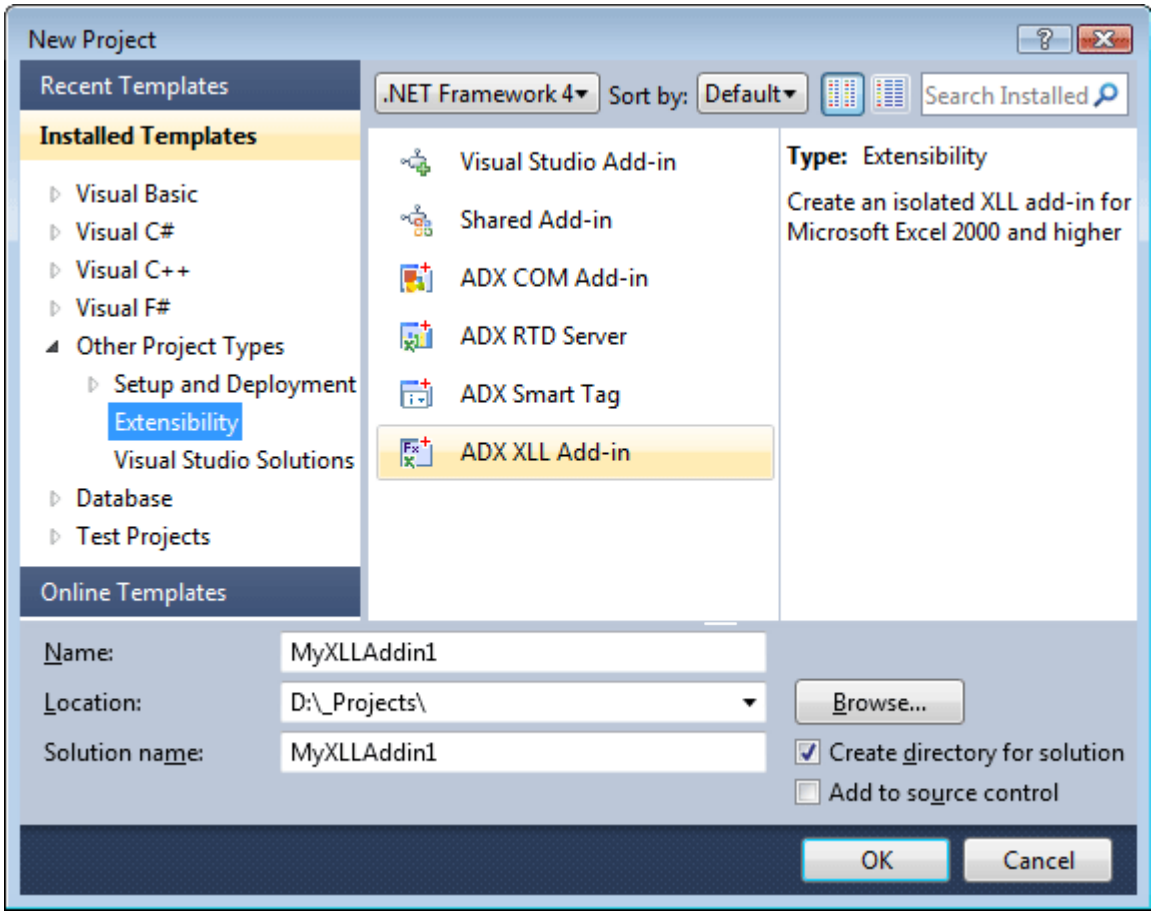
Create a setup project, build it, copy all setup files to the target PC and run the installer (see Deploying Office Extensions). See also Deploying Add-in Express Projects and Excel UDFs.



Your First XLL add-in

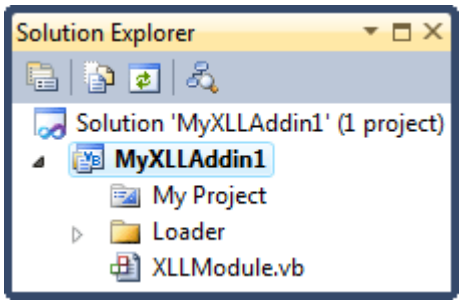
Step #1 – Creating a New Add-in Express XLL Add-in Project

Add-in Express adds the Add-in Express XLL Add-in project template to the Visual Studio IDE.



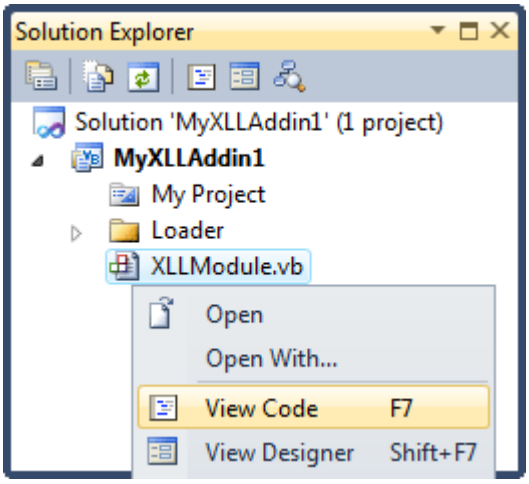
Click OK to start the XLL add-in project wizard. In the wizard window, you choose the programming language and interop assemblies to be used.

The wizard creates and opens a new solution containing the XLL add-in project.



The XLL add-in project contains the XLLModule.vb (or XLLModule.cs) file discussed in the next step.

Step #2 - Add-in Express XLL Module



The XLLModule.vb (or XLLModule.cs) file is the core part of the XLL add-in project. The XLL module is a container for Category components. It contains the XLLModule class, a descendant of ADXXLLModule that implements the interfaces required by the XLL technology and allows creating and configuring custom used defined functions (UDF). To review its source code, right-click the file in *Solution Explorer* and choose *View Code* in the context menu.

Please note the XLLContainer class in the code below. We describe its use in the next steps.

```
Imports System.Runtime.InteropServices
Imports System.ComponentModel

'Add-in Express XLL Add-in Module
<ComVisible(True)> _
Public Class XLLModule
    Inherits AddinExpress.MSO.ADXXLLModule

#Region " Component Designer generated code. "
   'Required by designer
   Private components As System.ComponentModel.IContainer

   'Required by designer - do not modify
   'the following method
   Private Sub InitializeComponent()
      '
      'XLLModule
      '
      Me.AddinName = "MyXLLAddin1"

   End Sub

#End Region

#Region " Add-in Express automatic code "

    'Required by Add-in Express - do not modify
    'the methods within this region

    Public Overrides Function GetContainer() As _
           System.ComponentModel.IContainer
       If components Is Nothing Then
```

```
            components = New System.ComponentModel.Container
        End If
        GetContainer = components
    End Function

    <ComRegisterFunctionAttribute()> _
    Public Shared Sub RegisterXLL(ByVal t As Type)
        AddinExpress.MSO.ADXXLLModule.RegisterXLLInternal(t)
    End Sub

    <ComUnregisterFunctionAttribute()> _
    Public Shared Sub UnregisterXLL(ByVal t As Type)
        AddinExpress.MSO.ADXXLLModule.UnregisterXLLInternal(t)
    End Sub

#End Region

#Region " Define your UDFs in this section "

    Friend Class XLLContainer

        Friend Shared ReadOnly Property _Module() As MyXLLAddin1.XLLModule
            Get
                Return CType(AddinExpress.MSO.ADXXLLModule. _
                    CurrentInstance, MyXLLAddin1.XLLModule)
            End Get
        End Property

    End Class

#End Region

    Public Sub New()
        MyBase.New()

        'This call is required by the Component Designer
        InitializeComponent()

        'Add any initialization after the InitializeComponent() call

    End Sub

    Public ReadOnly Property ExcelApp() As Excel._Application
        Get
            Return CType(HostApplication, Excel._Application)
        End Get
```

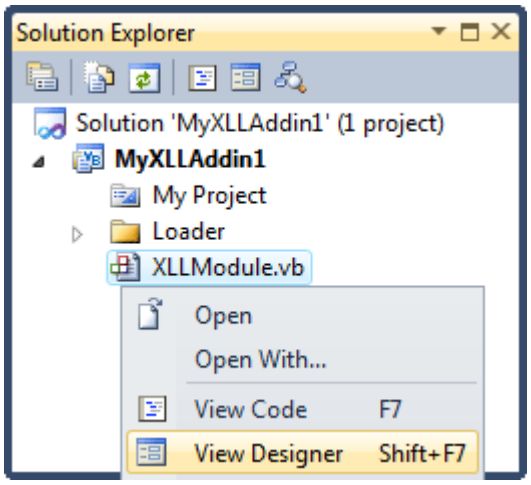
```
    End Property

End Class
```

Step #3 – Creating a New User-Defined Function

Just add a new `Public Shared` (`public static` in C#) function to the XLLContainer class. In this sample, we uncomment the *AllSupportedExcelTypes* function that demonstrates all the types that Excel can pass to user-defined functions.

```
Friend Class XLLContainer

...

   Public Shared Function AllSupportedExcelTypes(ByVal arg As Object) _
         As String
      If (TypeOf arg Is Double) Then
         Return "Double: " + arg.ToString()
      ElseIf (TypeOf arg Is String) Then
         Return "String: " + arg
      ElseIf (TypeOf arg Is Boolean) Then
         Return "Boolean: " + arg.ToString()
      ElseIf (TypeOf arg Is AddinExpress.MSO.ADXExcelError) Then
         Return "ExcelError: " + arg.ToString()
      ElseIf (TypeOf arg Is Object(,)) Then
         Return String.Format("Array[{0},{1}]", arg.GetLength(0), _
                arg.GetLength(1))
      ElseIf (TypeOf arg Is System.Reflection.Missing) Then
         Return "Missing"
      ElseIf (arg Is Nothing) Then
         Return "Empty"
      ElseIf (TypeOf arg Is AddinExpress.MSO.ADXExcelRef) Then
         Return "Reference: " + arg.ConvertToA1Style()
      ElseIf (TypeOf arg Is AddinExpress.MSO.ADXExcelRef) Then
         Return String.Format("Reference [{0},{1},{2},{3}]", _
             arg.ColumnFirst, arg.RowFirst, arg.ColumnLast, arg.RowLast)
      Else
         Return "Unknown Type"
      End If
   End Function

End Class
```

Step #4 - Configuring UDFs



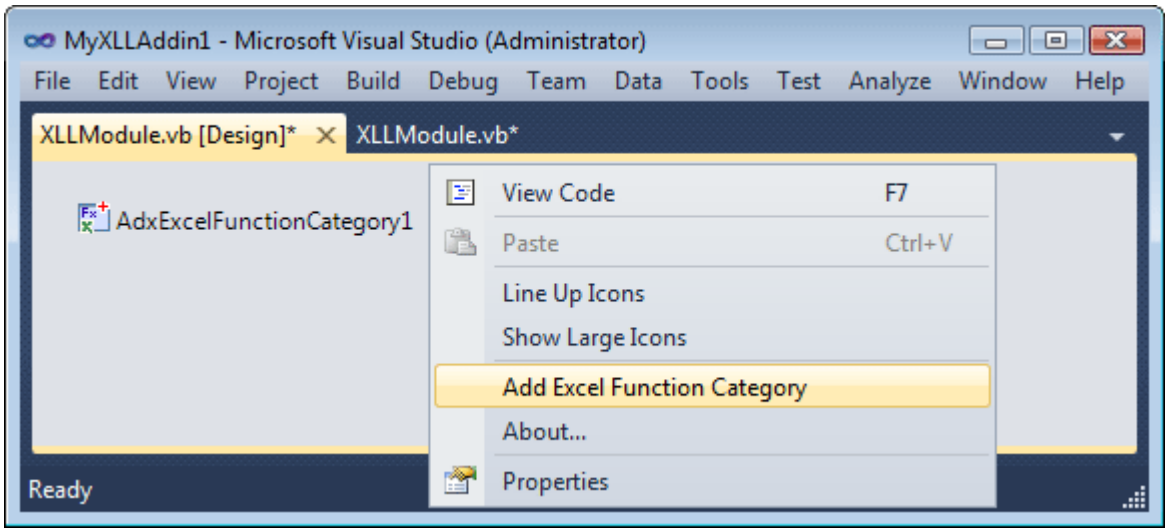
To integrate the XLL add-in in Excel, you have to supply Excel with a user-friendly add-in name, function names, parameter names, help topics, etc.

When developing Add-in Express XLL add-ins, you start with adding a custom function. Then you create an Excel function category, add a function descriptor and bind the function to the function descriptor. Finally, you add parameter descriptors to the function descriptor. Let's go.

In *Solution Explorer*, right-click the XLL module and choose *View Designer* in the popup menu.

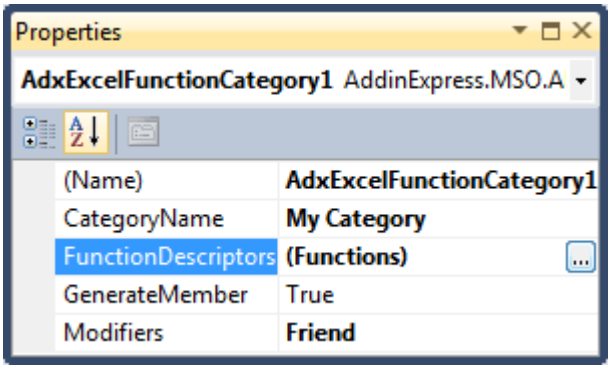
Specify the add-in name in the *Properties* window.

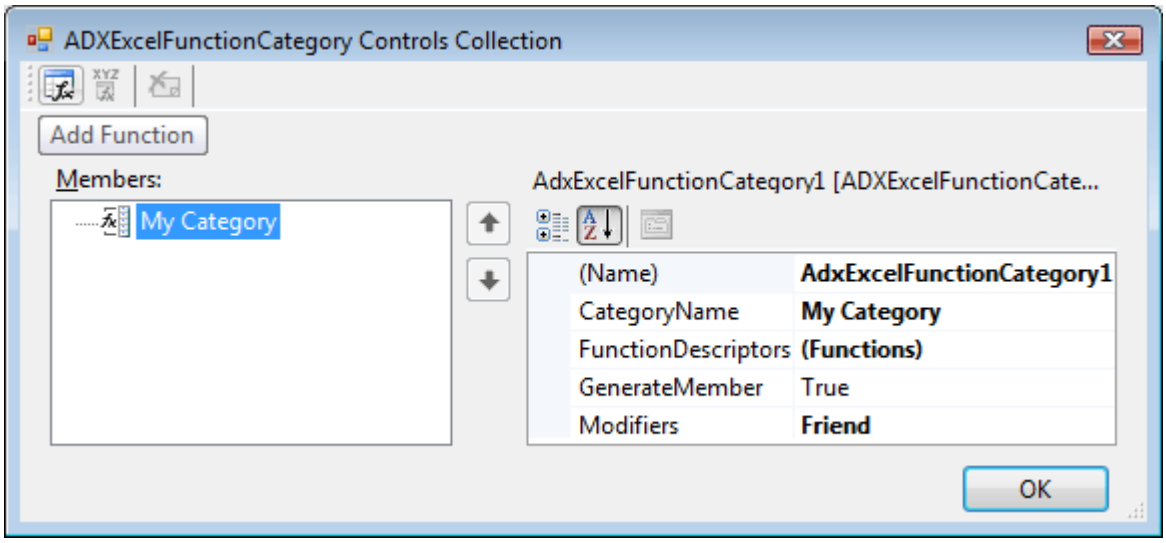
Now you right-click the designer surface and, in the popup menu, choose *Add Excel Function Category*.



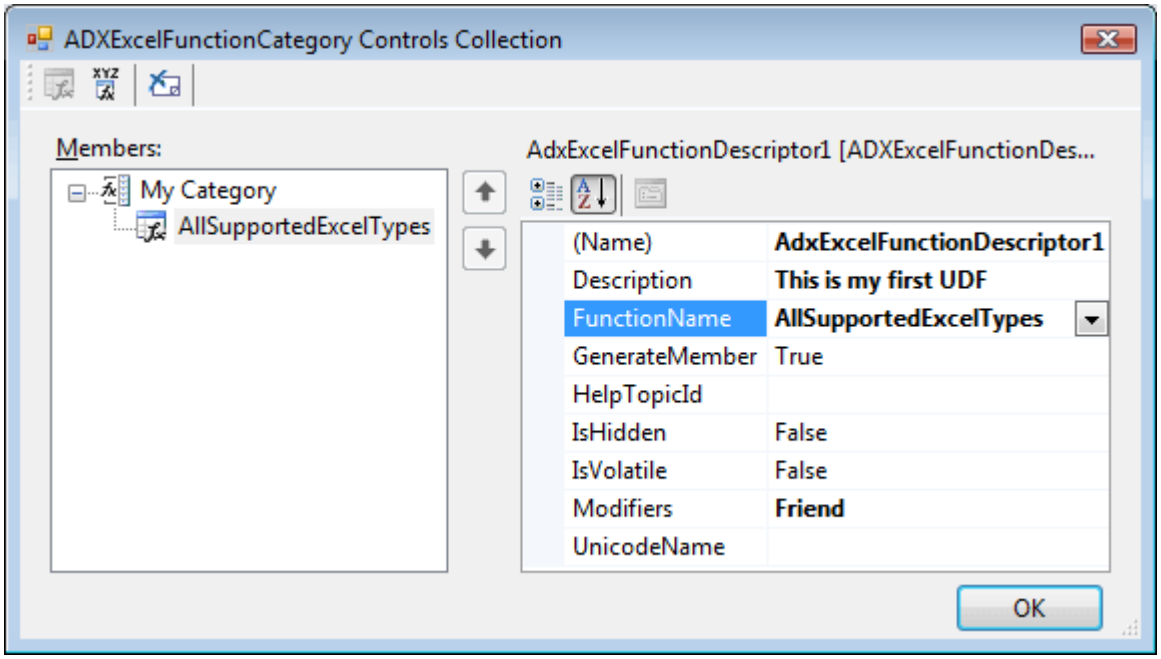
This adds an Excel Function Category component onto the XLL module.

Select the component, specify the category name and start the property editor of the Functions property of the ADXExcelFunctionCategory component.



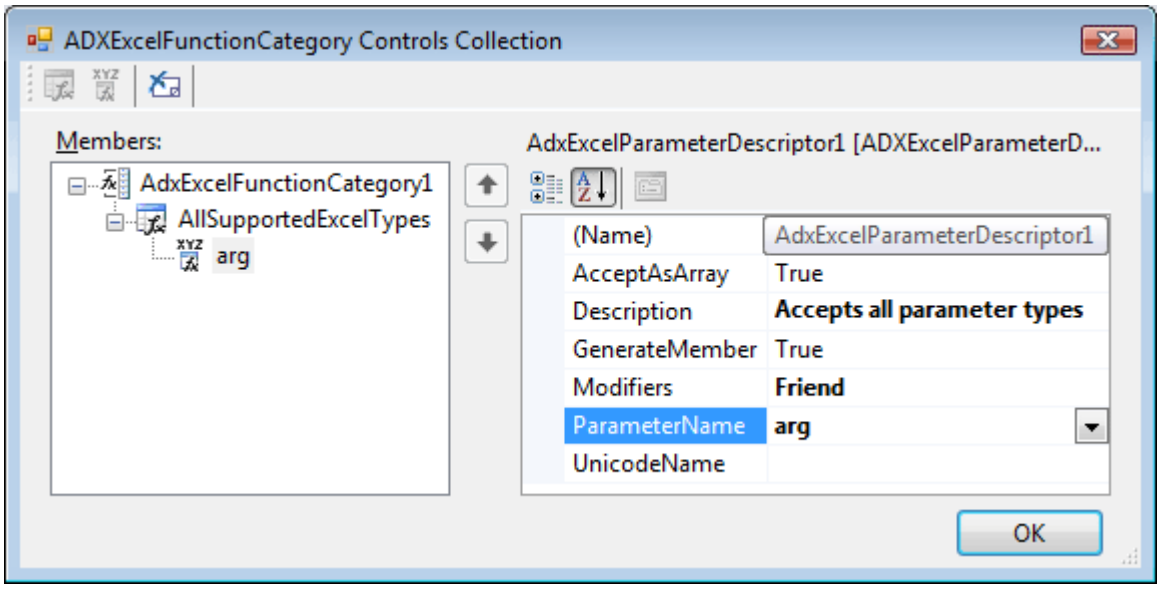


Now, you add a function descriptor and bind it to the function you created in the XLLContainer class.



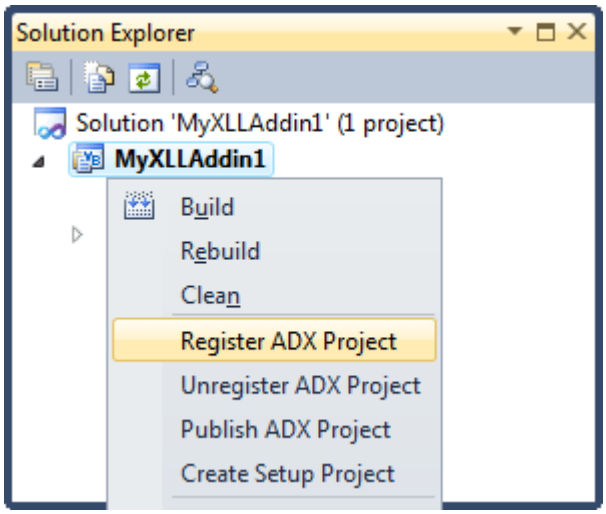
For a function descriptor, the properties of interest are FunctionName and IsVolatile. The former is a combo box that allows choosing a function from the list of functions defined in the XLLContainer class. As to the latter, if it is set to true, Excel will call the appropriate function whenever it recalculates the worksheet.

In the same way, you describe the arguments of the function: add a parameter descriptor and select a parameter in the ParameterName property (see below).



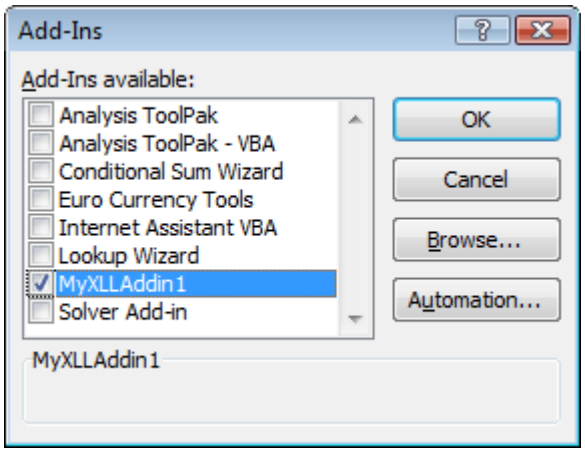
When renaming functions and arguments, you have to reflect these changes in appropriate descriptors. In the opposite case, Excel will not receive the information required.

Step #5 – Running Your XLL Add-in

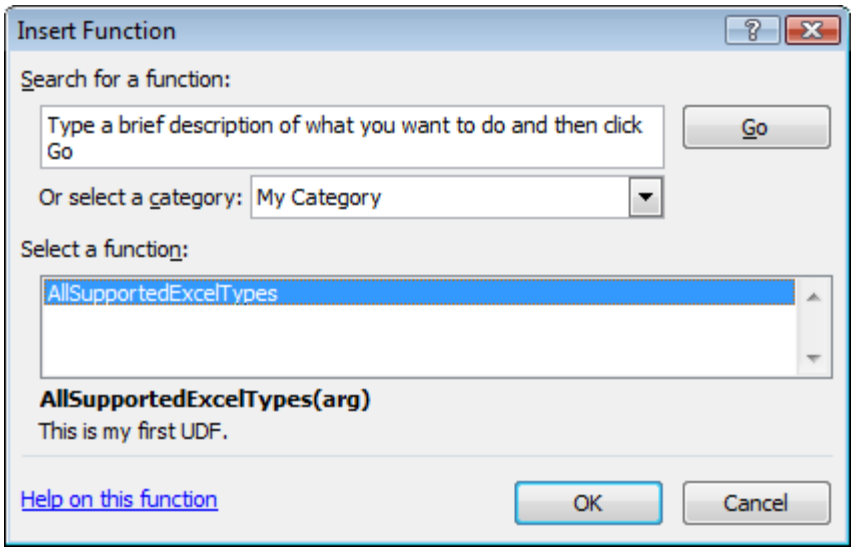


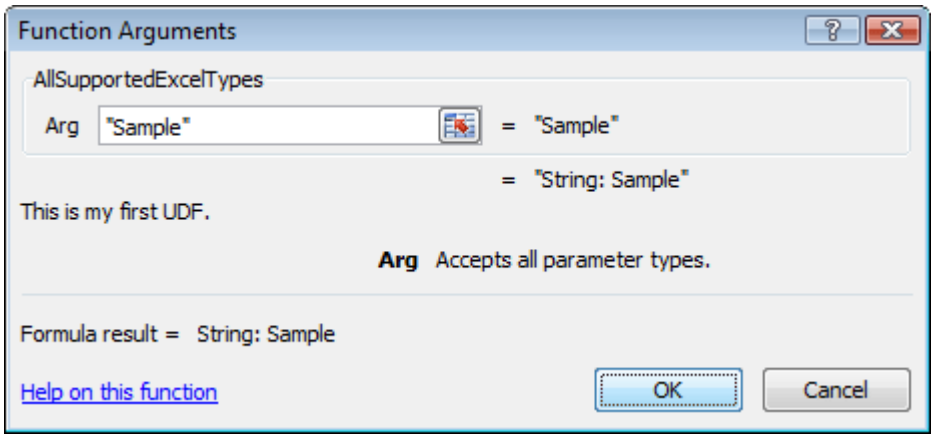
Build and register your XLL add-in, restart Excel, and check if your add-in works.

First, find it in the *Add-ins* dialog: see *Tools* | *Add-ins* in Excel 2000-2003, *Office Button* | *Excel Options* | *Add-ins* | *Manage "Excel add-ins"* | *Go* in Excel 2007 and *File* | *Options* | *Add-ins* | *Manage "Excel add-ins"* | *Go* in Excel 2010.



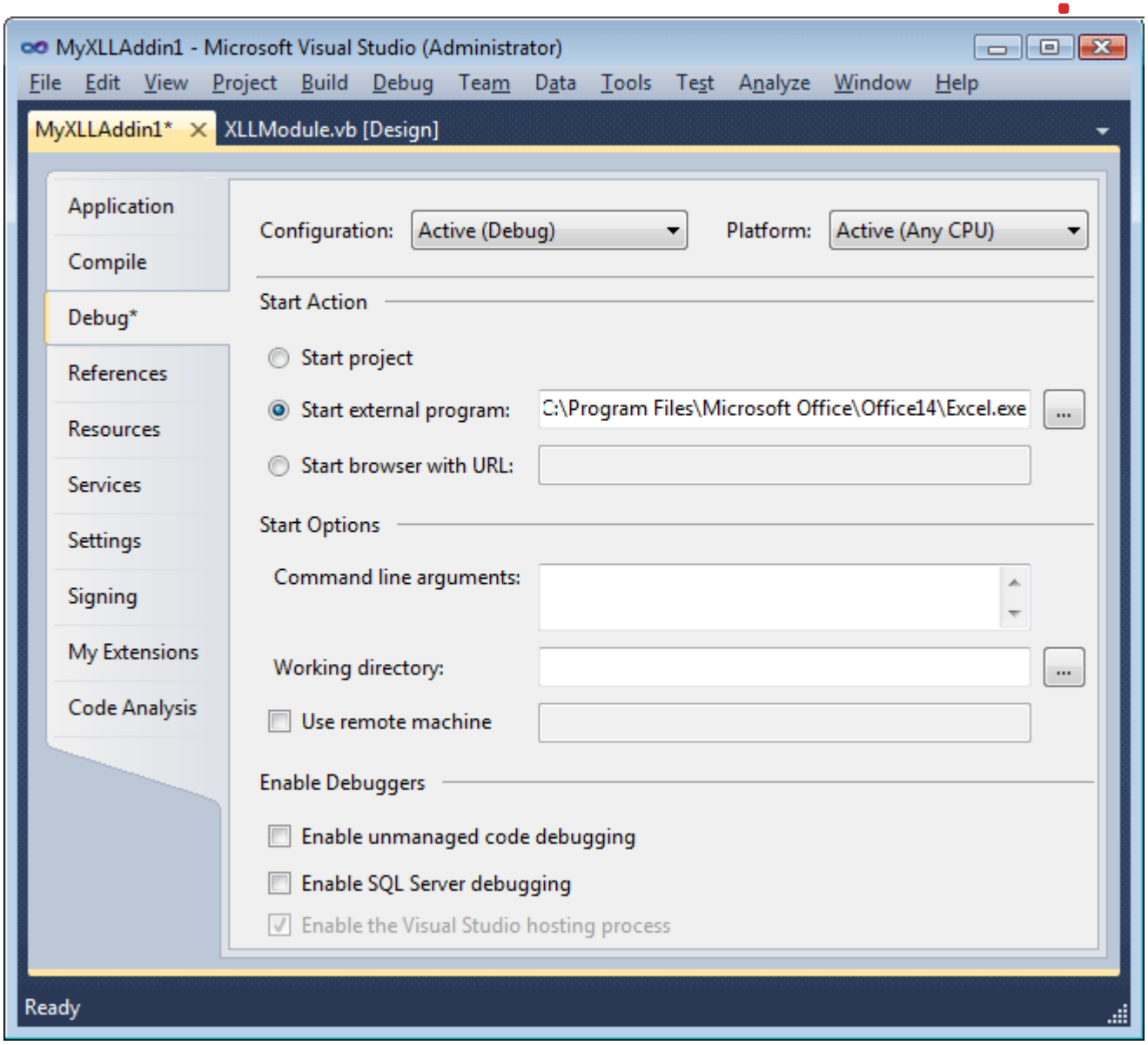
Now you can use your UDF in the Insert Function wizard:





Step #6 – Debugging the XLL Add-in

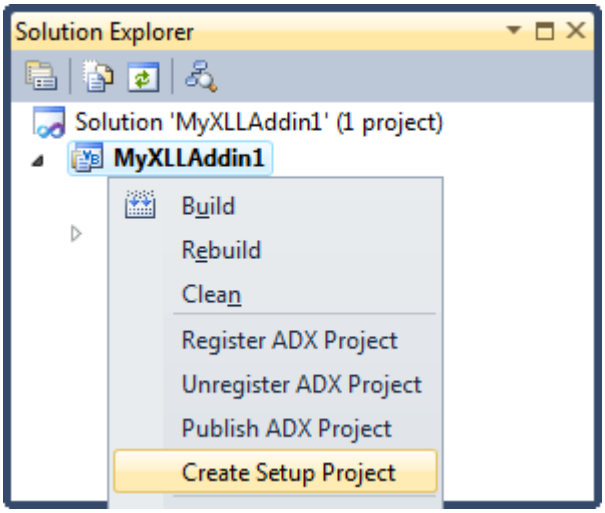
To debug your add-in, in the Project Options window, specify the full path to excel.exe in Start External Program and run the project.



Step #7 - Deploying the XLL Add-in

Create a setup project, build it, copy all setup files to the target PC and run the installer (see Deploying Office Extensions).

See also Deploying Add-in Express Projects and Excel UDFs.



How Your Office Extension Loads Into an Office Application

Registry Keys

Any Office extension – a COM add-in, Excel add-in, RTD server, or smart tag – must be installed and registered because Office looks for extensions in the registry. In other words, to get your add-in to work, 1) add-in files must be installed to a location accessible by the add-in users and 2) registry keys must be created that specify which Office application will load the add-in and which PC users may use the add-in. The necessity to create registry keys is the reason why **you cannot use XCOPY deployment** for a COM add-in, Excel XLL add-in, RTD server, or Smart tag.

Although Add-in Express creates all registry keys for you, you might need to find the keys when debugging your add-ins. The main intention of this section is to provide you with information on this.

Locating COM Add-ins in the Registry

Depending on the value of the RegisterForAllUsers property of the add-in module, the main registry entry of a COM add-in is located at:

```
{HKLM or HKCU}\Software\Microsoft\Office\{host}\AddIns\{your add-in ProgID}
```

If the RegisterForAllUsers property of the add-in module is true, the add-in is registered in HKEY_LOCAL_MACHINE, otherwise the key is located in HKEY_CURRENT_USER.

Pay attention to the *LoadBehavior* value defined in this key; typically, it is 3. If *LoadBehavior* is 2 when your run your add-in, this may be an indication of an unhandled exception at add-in startup.

The registry key above notifies the corresponding Office application that there's an add-in to load.

FYI, the COM add-in is a COM object registered in

```
HKEY_CLASSES_ROOT\CLSID\{Add-in Express Project GUID}
```

Locating Excel UDF Add-ins in the Registry

Registering a UDF adds a value to the following key:

```
HKEY_CURRENT_USER\Software\Microsoft\Office\{Office version}.0\Excel\Options
```

The value name is OPEN or OPEN{n} where n is 1, if another UDF is registered, 2 - if there are two other XLLs registered, etc. The value contains a string, which is constructed in the following way:

```
str = "/R " + "" + pathToTheDll + ""
```

Add-in Express Loader

All Office applications are unmanaged while all Add-in Express based add-ins are managed class libraries. Therefore, there must be some software located between Office applications and your add-ins. Otherwise, Office applications will not know of your .NET add-ins and other Office extensions. That software is called a shim. Shims are unmanaged DLLs that isolate your add-ins in a separate application domain.

When you install your add-in, the registry settings for the add-in will point to the shim. And the shim will be the first DLL examined by the host application when it starts loading your add-in or smart tag.

Add-in Express provides the shim of its own, called Add-in Express loader. The loader (adxloader.dll, adxloader64.dll) is a compiled shim not bound to any certain Add-in Express project. Instead, the loader uses the adxloader.dll.manifest file containing a list of .NET assemblies that must be registered. The loader's files (adxloader.dll, adxloader64.dll and adxloader.dll.manifest) must always be located in the Loader subdirectory of the Add-in Express project folder. When a project is being rebuilt or registered, the loader files are copied to the project's output directory. You can sign the loader with a digital signature and, in this way, create trusted COM extensions for Office.

Add-in Express Loader Manifest

The manifest (adxloader.dll.manifest) is the source of configuration information for the loader. Below, you see the content of a sample manifest file.

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <assemblyIdentity name="MyAddin14, PublicKeyToken=f9f39773da5c568a" />
  <loaderSettings generateLogFile="true" shadowCopyEnabled="true"
privileges="user"> configFileName="app.config"
  <logFileLocation>C:\MyLog.txt</logFileLocation>
 </loaderSettings>
</configuration>
```

The manifest file allows generating the log file containing useful information about errors on the add-in loading stage. The default location of the log file is {user profile}\Documents\Add-in Express\adxloader.log. You can change the location using the `logFileLocation` node; relative paths are also acceptable. The manifest file allows you to disable the Shadow Copy feature of the Add-in Express loader, which is enabled by default (see Deploying – Shadow Copy). The `privileges` attribute accepts the `"user"` string indicating that the Add-in Express based setup projects can be run with non-administrator privileges. Please, note, any other value will require administrator privileges to install your project. You should be aware that the value of this attribute is controlled by the RegisterForAllUsers property value of add-in and RTD modules (see Add-in Express Basics). If RegisterForAllUsers is True and `privileges`="`user`", a standard user running the installer will be unable to install your Office extension. If RegisterForAllUsers is False and `privileges`="`administrator`", your Office extension will be installed for the administrator only.

In addition, you can run regsvr32 against the adxloader.dll. If a correct manifest file is located in the same folder, this will register all Add-in Express projects listed in the loader manifest.

How the Loader Works

Consider an ideal case, when all required files are supplied, registry keys are created and permissions are correct. If so, when the Office application discovers an appropriate registry key (see Locating COM Add-ins in the Registry), it finds the loader's DLL, loads it and calls a method implemented by the loader in accordance with COM rules. The loader initializes CLR (Common Language Runtime), reads the manifest, creates an AppDomain, loads your assembly into the domain, and creates an instance of your add-in module (this runs the constructor of the module). Then the loader generates the AddinInitialize and AddinStartupComplete events of the module, connects the module to events of the host application and waits for the event that specifies the end of the job. When such an event occurs, the loader disconnects the module from the host application events, and generates the AddinBeginShutdown and AddinFinalize events of the module (see also Custom Actions When Your COM Add-in Is Uninstalled).

Loader's Log

If the manifest requires creating a log file (see the `generateLogFile` attribute at Add-in Express Loader Manifest), the log file is created in the location specified by the manifest or in {Documents}\Add-in Express\adxloader.log (default).

Note that the log is re-created whenever you install/uninstall the add-in and when the Office application loads it.

Deploying Add-in Express Projects

Updatability of Office extensions

Add-in Express supports two schemes of updating an Office extension: ClickOnce Deployment and Web-based MSI deployment (also known as ClickTwice :). These schemes differ in user permissions required for installing and updating an add-in: while ClickOnce requires non-admin permissions, ClcikTwice :) allows both admin and non-admin permissions.

Note that updating an Office extension requires restarting the host application(s). This occurs because there is no way to unload an Office extension other than by closing the host application.

How to Find Files on the Target Machine Programmatically?

You can find the actual location of your files on the target PC using the following code:

```
System.Reflection.Assembly.GetExecutingAssembly().CodeBase
```

Files to Deploy

The tables below contain minimal sets of files required for your Office extension to run.

Office add-ins, XLL add-ins

File name	Description
AddinExpress.MSO.2005.dll	Command bar and Ribbon controls, COM add-in and XLL
Interop assemblies	All interops required for your add-in
extensibility.dll	Contains the definition of the IDTExtensibility2 COM interface
adxloader.dll	32-bit loader; required for Office 2000-2007, and Office 2010 32-bit
adxloader64.dll	64-bit loader; required for Office 2010 64-bit
adxloader.dll.manifest	Loader manifest
adxregistrator.exe	Add-in registrator

For an XLL add-in, the loader names include the assembly name, say, adxloader.MyXLLAddin1.dll (adxloader64.MyXLLAddin1.dll).

Excel Automation add-ins

File name	Description
AddinExpress.MSO.2005.dll	Excel automation add-ins
Interop assemblies	All interops required for your add-in
extensibility.dll	Contains the definition of the IDTExtensibility2 COM interface
adxregistrator.exe	The add-in registrator

RTD servers

File name	Description
AddinExpress.RTD.2005.dll	Excel RTD Server
adxloader.dll	32-bit loader; required for Office 2000-2007, and Office 2010 32-bit
adxloader64.dll	64-bit loader; required for Office 2010 64-bit
adxloader.dll.manifest	Loader manifest
adxregistrator.exe	Add-in registrator

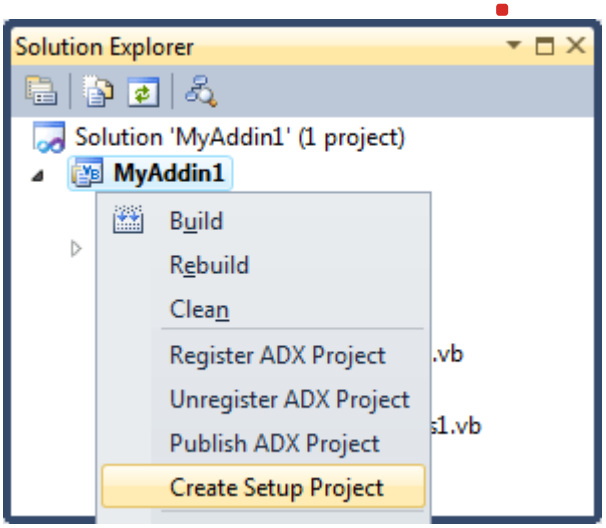
Smart tags

File name	Description
AddinExpress.SmartTag.2005.dll	Smart Tag
adxloader.dll	32-bit loader; required for Office 2000-2007, and Office 2010 32-bit
adxloader64.dll	64-bit loader; required for Office 2010 64-bit
adxloader.dll.manifest	Loader manifest
adxregistrator.exe	Add-in registrator

Web-based MSI deployment

This feature is currently described in a series of articles posted on our blog. The initial post is Add-in Express 2010 MSI-based web deployment – ClickTwice :).

Creating Setup Projects in Visual Studio



To help you create an installer for your Office extension, Add-in Express provides the setup project wizard accessible via the *Create Setup Project* item in the context menu of the project item in the *Solution Explorer window.*

Creating Setup Projects Manually

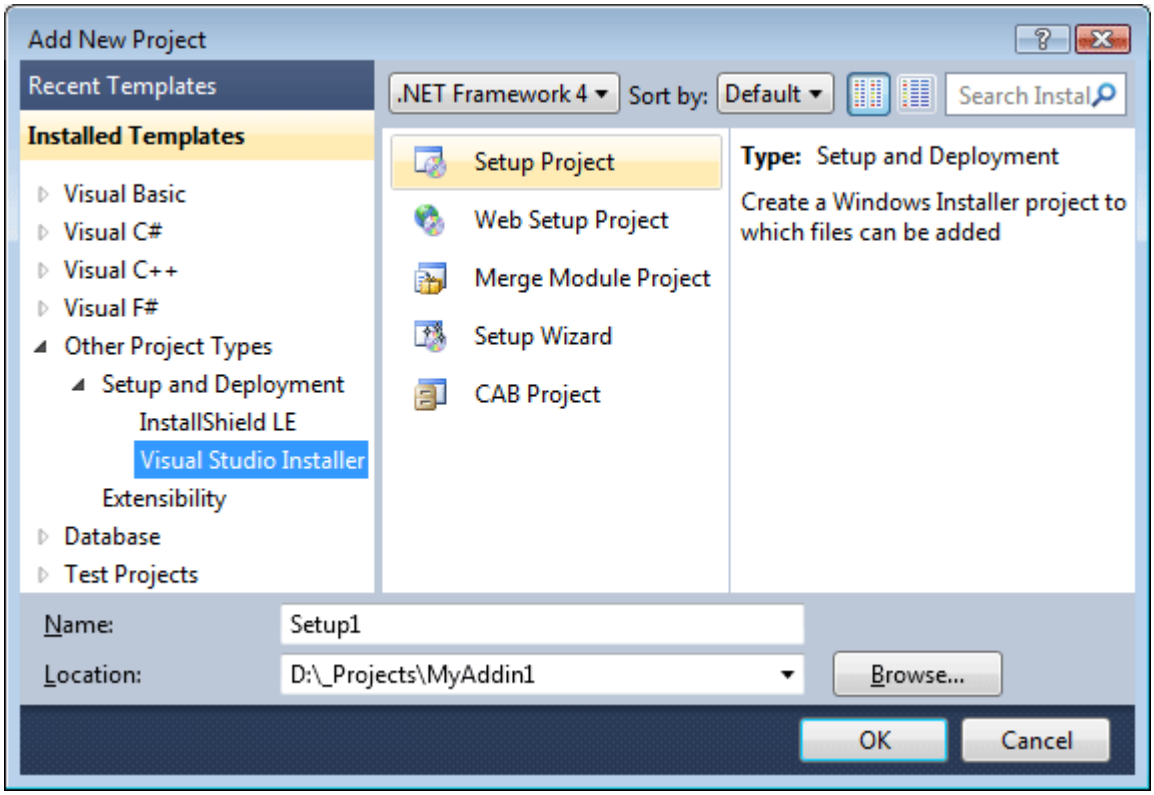
Note that you can create a setup project using the setup project wizard and check all the below-mentioned settings.

To create a setup project manually, follow the steps below.

- **Add a New Setup Project**

Right-click the solution item and choose Add | New Project.

In the Add New Project dialog, select the Setup Project item and click OK. This will add the setup project to your solution.



In the Add New Project dialog, select the Setup Project item and click OK. This will add the setup project to your solution.

- **File System Editor**

Right-click the setup project item and choose *View* | *File System* in the context menu.

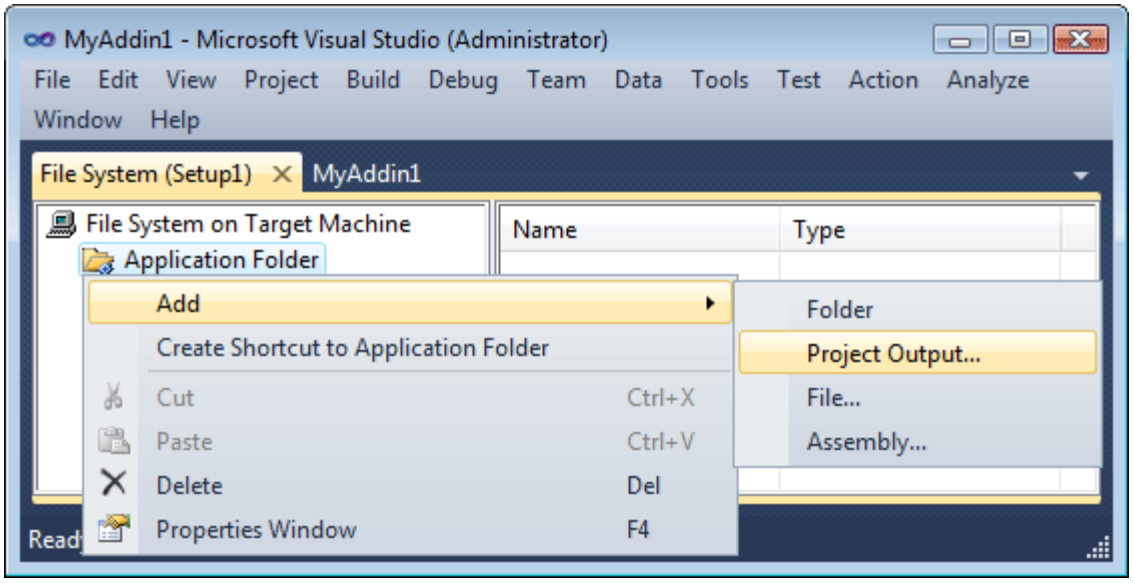
- **Application Folder \ Default Location**

Select the Application Folder and specify its DefaultLocation property as follows:

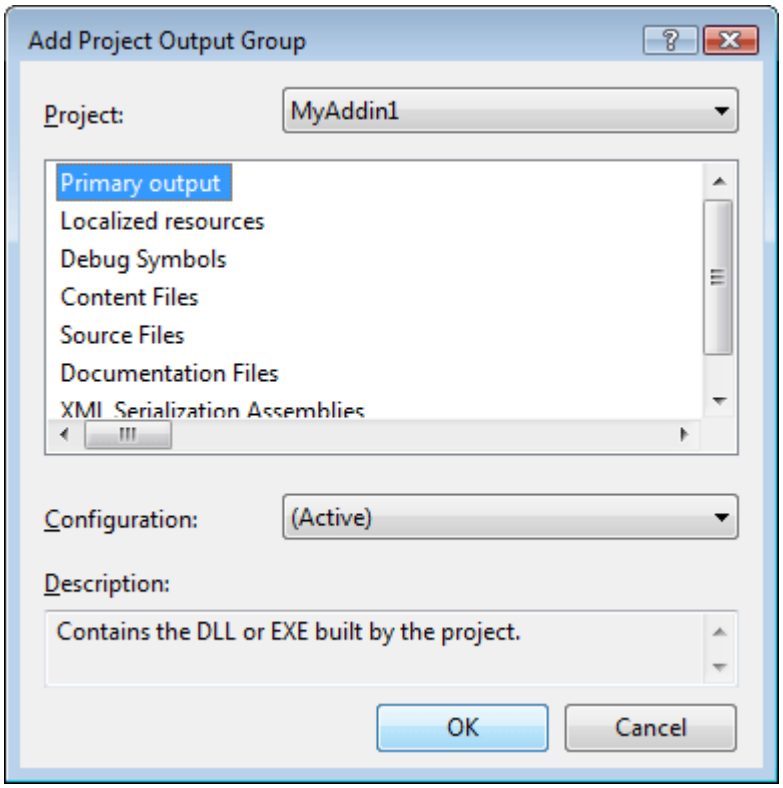
- If the RegisterForAllUsers property of the module is true, set DefaultLocation = [ProgramFilesFolder][Manufacturer]\[ProductName]
- If the RegisterForAllUsers property of the module is false or, if you deploy a smart tag or Excel UDF, set DefaultLocation = [AppDataFolder][Manufacturer]\[ProductName]

- **Primary Output**

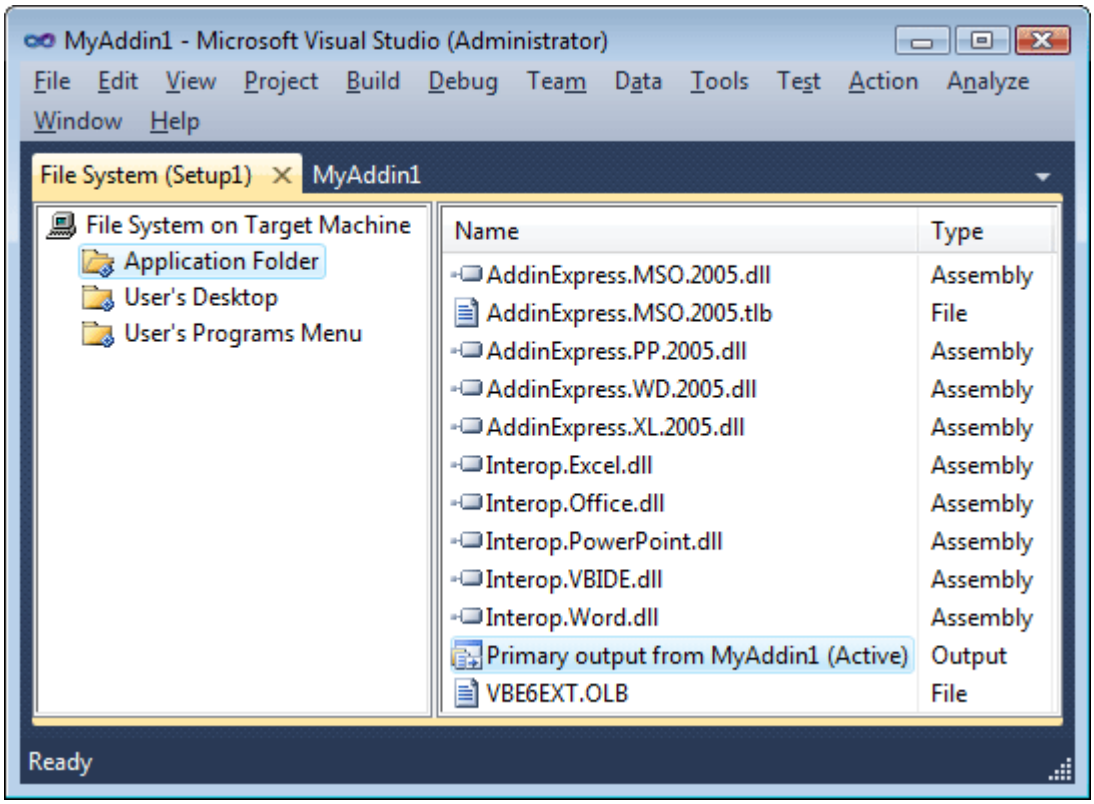
Right-click the Application Folder item and choose Add | Project Output



In the *Add Project Output Group* dialog, select the *Primary output* item of your Add-in Express project and click OK.

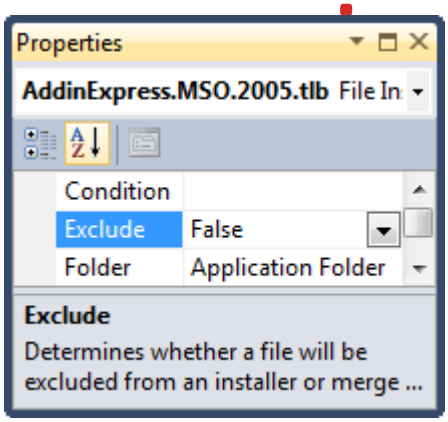


For the add-in described in Your First Microsoft Office COM Add-in, this adds the following entries to the Application Folder of the setup project:



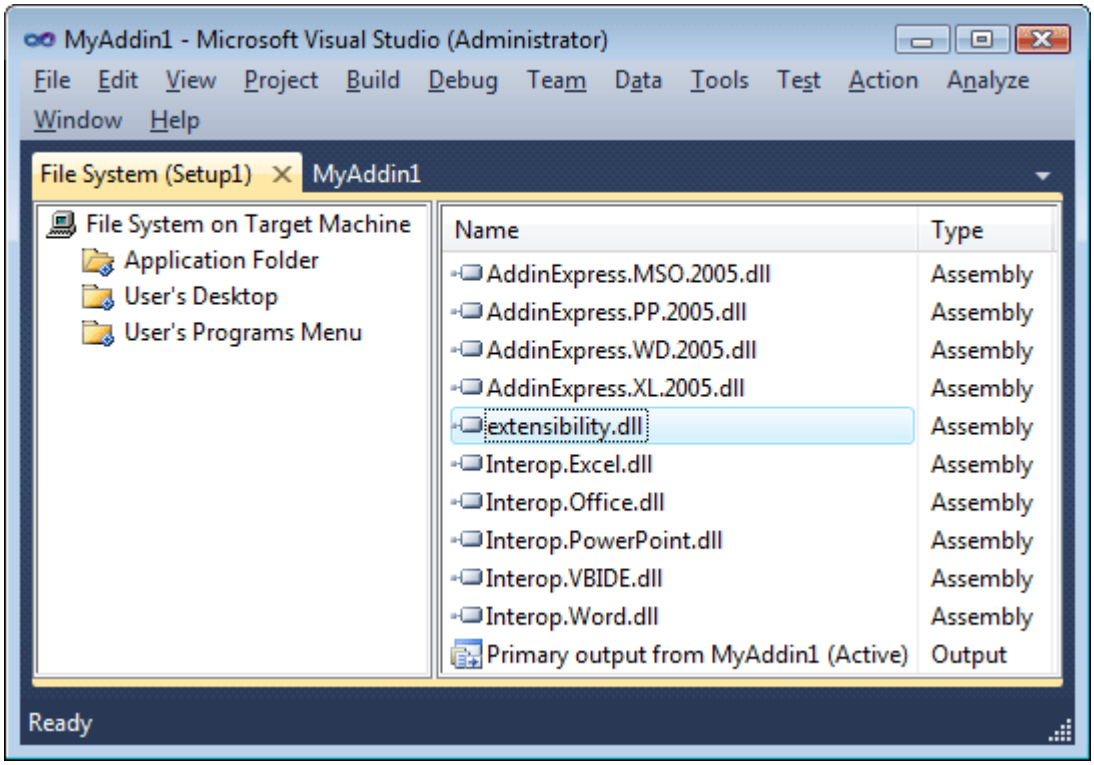
Select AddinExpress.MSO.2005.tlb and, in the Properties window, set the Exclude property to true. If you use version-neutral interops, please exclude the VB6EXT.OLB file in the same way.

Always exclude all .TLB and .OLB files from the setup project except for .TLBs that you create yourself.



- **Extensibility.dll**

If Extensibility.dll isn't listed in the *Detected Dependencies* section of the setup project, locate the file in the {Add-in Express}\Bin and add it to the Application Folder of the setup project.



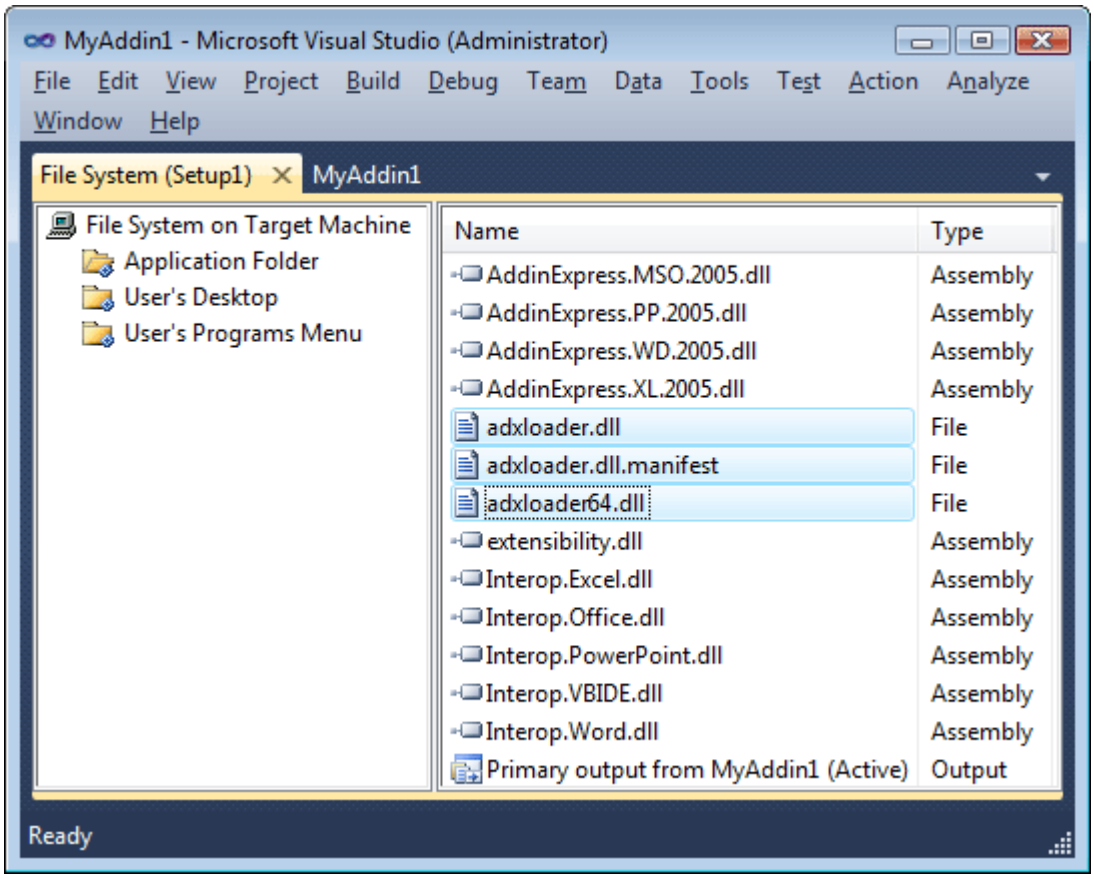
- **Project-depended Resources**

Now you add all resources (e.g. assemblies, DLLs or any resources) required for your project.

- **Add-in Express Loader and Manifest**

Add adxloader.dll, adxloader64.dll and adxloader.dll.manifest files from the Loader folder of the add-in project directory to the Application Folder.

For an XLL add-in, the loader names include the assembly name, say, adxloader.MyXLLAddin1.dll.

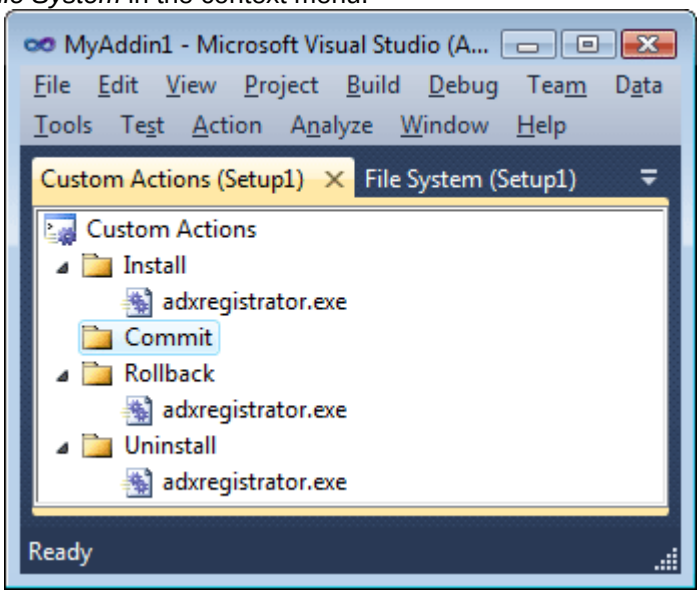


- **Add-in Express Registrator**

Add {Add-in Express}\Redistributables\adxregistrator.exe to the Application Folder.

- **Custom Actions Editor**

Right-click the setup project item and choose *View* | *File System* in the context menu.



- **Add Custom Actions**

Add a new action to the *Install*, *Rollback*, *Uninstall* sections. Use adxregistrator.exe as an item for the custom actions.

➢ Custom Actions Arguments

Add the strings below to the Arguments properties of the following custom actions:

- Install

```
/install="{assembly name}.dll" /privileges={user OR admin}
```

- Rollback

```
/uninstall="{assembly name}.dll" /privileges={user OR admin}
```

- Uninstall

```
/uninstall="{assembly name}.dll" /privileges={user OR admin}
```

If a COM add-in or RTD server is installed on the per-user basis, or if you deploy a smart tag or an Excel UDF, the value of the privileges argument above is `user`. If a COM add-in or RTD server is installed on the per-machine basis, in other words, if the RegisterForAllUsers property of the corresponding module is true, the value of the privileges argument above is `administrator`.

Say, for an add-in described in Your First Microsoft Office COM Add-in, the Arguments property for the *Install* custom action contains the following string:

```
/install="MyAddin1.dll" /privileges=user
```

➢ Dependencies

Right click on the *Detected Dependencies* section of the setup project and choose *Refresh Dependencies* in the context menu. Also, exclude all dependencies that are not required for your setup.

➢ Launch Conditions

Right-click the setup project item and choose *View* | *Launch Conditions* in the context menu.

Make sure that the *.NET Framework* launch condition specifies a correct .NET Framework version and correct download URL. Note that we recommend using launch conditions rather than pre-requisites because installing a pre-requisite usually requires administrative permissions and in this way installing a per-user Office extension may result in installing the extension for the administrator, but not for the user who ran the installer.

➢ Prerequisites

Right click the setup project and open the *Properties* dialog.

If administrative permissions are required to install prerequisites, then for a per-user Office extension, the elevation dialog will be shown on UAC-enabled systems. If the administrator's credentials are entered in this situation, then the installer will be run on behalf of the administrator and therefore, the Office extension will be installed for the administrator, not for the user who originally ran the installer.

Click the *Prerequisites* button and, in the *Prerequisites* dialog, select required prerequisites.

➢ **The Final Touch**

Rebuild the setup project. Specify the following command line in the PostBuildEvent property of the setup project:

- If the RegisterForAllUsersProperty of the module is false or if that property is missing:

```
{Add-in Express}\Bin\adxPatch.exe %BuiltOuputPath% /UAC=Off
```

- If the RegisterForAllUsersProperty of the module is true:

```
{Add-in Express}\Bin\adxPatch.exe %BuiltOuputPath% /UAC=On
```

Now build the setup project, copy all setup files to the target PC and run the .msi file to install the add-in. However, to install pre-requisites, you will need to run setup.exe.

ClickOnce Deployment

ClickOnce Overview

What follows below is a brief compilation of the following Internet resources:

- ClickOnce article from Wikipedia
- ClickOnce FAQ on windowsclient.net
- Introduction to ClickOnce deployment on msdn2.microsoft.com (also compares ClickOnce and MSI)
- ClickOnce Deployment in .NET Framework 2.0 on 15seconds.com

ClickOnce is a deployment technology introduced in .NET Framework 2.0. Targeted to non-administrator-privileges installations it also allows updating your applications. Subject to many restrictions, it isn't a panacea in any way. Say, if your prerequisites include .NET Framework 2.0 and the user doesn't have it installed, your application (as well as an add-in) will not be installed without administrator privileges. In addition, ClickOnce will not allow installing shared components, such as custom libraries. It is quite natural, though.

When applied to a Windows forms application, ClickOnce deployment implies the following steps:

- Publishing an application

You deploy the application to either File System (CD/DVD included) or Web Site. The files include all application files as well as application manifest and deployment manifest. The application manifest describes the application itself, including the assemblies, dependencies and files that make up the application, required permissions, and the location where updates will be available. The deployment manifest describes how the

application is deployed, including the location of the application manifest, and the version of the application that the user should run. The deployment manifest also contains an update location (a Web page or network file share) where the application checks for updated versions. ClickOnce Publish properties are used to specify when and how often the application should check for updates. Update behavior can be specified in the deployment manifest, or it can be presented as user choices in the application's user interface by means of the ClickOnce API. In addition, Publish properties can be employed to make updates mandatory or to roll back to an earlier version.

- Installing the application

The user clicks a link to the deployment manifest on a web page, or double-clicks the deployment manifest file in Windows Explorer. In most cases, the end user is presented with a simple dialog box asking the user to confirm installation, after which installation proceeds and the application is launched without further intervention. In cases where the application requires elevated permissions, the dialog box also asks the user to grant permission before the installation can continue. This adds a shortcut icon to the Start menu and lists the application in the Control Panel/Add Remove Programs. Note, it does not add anything to the registry, the desktop, or to Program Files. Note also that the application is installed into the ClickOnce Application Cache (per user).

- Updating the application

When the application developer creates an updated version of the application, they also generate a new application manifest and copy files to a deployment location—usually a sibling folder to the original application deployment folder. The administrator updates the deployment manifest to point to the location of the new version of the application. When the user opens the deployment manifest, the ClickOnce loader runs it and in this way, the application is updated.

Add-in Express ClickOnce Solution

Add-in Express adds the *Publish Add-in Express Project* item to the Build menu in Visual Studio 2005, 2008 and 2010. When you choose this item, Add-in Express shows the Publish dialog that generates the deployment manifest and places it into the Publish subfolder of the solution folder. In addition, the dialog generates the application manifest and places it to the Publish / <AssemblyVersion> folder. Then the dialog copies the add-in files and dependencies (as well as the Add-in Express loader and its manifest) to the same folder.

One more file copied to the Publish / <AssemblyVersion> folder is called the Add-in Express Launcher for ClickOnce Applications or the launcher. Its file name is adxlauncher.exe. This file is the heart of the Add-in Express ClickOnce Solution. The launcher is a true ClickOnce application. It will be installed on the user's PC and listed in the Start menu and Add / Remove Programs. The launcher registers and unregisters your add-in, and it provides a form that allows the user to register, unregister, and update your add-in. It also allows the user to switch between two latest versions of your add-in. Overall, the launcher takes upon itself the task of communicating with the ClickOnce API.

Notes

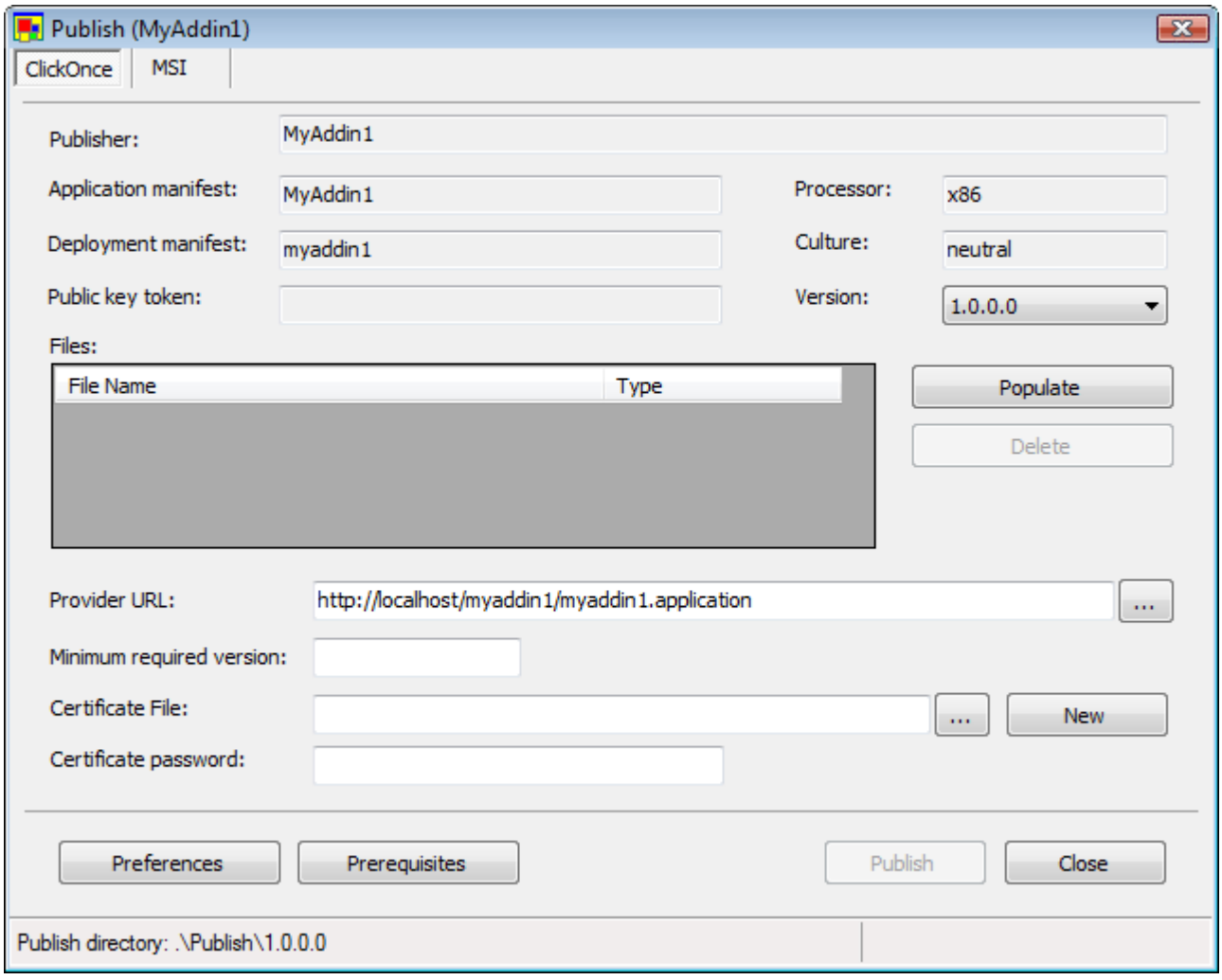
1. The launcher (adxlauncher.exe) is located in {Add-in Express}\Redistributables. You can check its properties (name, version, etc) in Windows Explorer. Subsequent releases will replace this file with its newer versions. And this may require you to copy a new launcher version to your Publish\<AssemblyVersion> folder.

2. For your convenience, we recommend avoiding using the asterisk in the <AssemblyVersion> tag.

All this will be done when you publish the add-in. However, let's click the *Publish Add-in Express Project* menu item to see the *Publish* dialog.

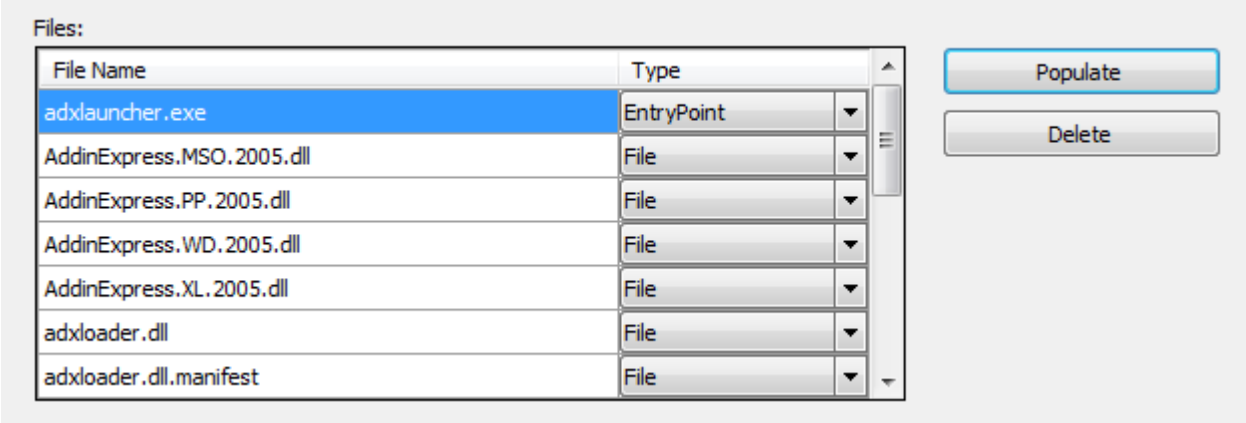
On the Development PC

The *Publish* dialog helps you create application and deployment manifests. In the current Add-in Express version, it shows the following form:



Step #1 – Populating the Application Manifest

Just click *Populate*. This is the moment when all the above-mentioned folders are created and files are copied.



To set a custom icon for the launcher, you can add a .ico file and mark it as Icon File in the *Type* column of the *Files* list box.

How to add extra files to the application manifest?

The current release does not provide the user interface for adding additional files and/or folders. However, you can copy the files and/or folders required by your add-in to the Publish / <AssemblyVersion> folder and click the Populate button again.

Step #2 – Specifying the Deployment / Update Location

You fill the Provider URL textbox with the URL of the deployment manifest (remember, it is located in the Publish folder). For Web-site based deployment, the format of the URL string is as follows:

```
http://<web-site path>/<deployment manifest name>.application
```

Please note that <deployment manifest name> must be entered in lower case. You can copy it from the Deployment manifest textbox in the Publish dialog window.

When debugging, you can create a Virtual Directory on your IIS server and bind it to the folder where your deployment manifest is located (the Publish folder is the easiest choice). In this case, the Provider URL could be like this:

```
http://localhost/clickoncetest/myclickonceaddin1.application
```

When releasing a real add-in, the Provider URL must specify the location of the next update for the current add-in version. You can upload version 1.0 of your add-in to any web or LAN location and specify the update

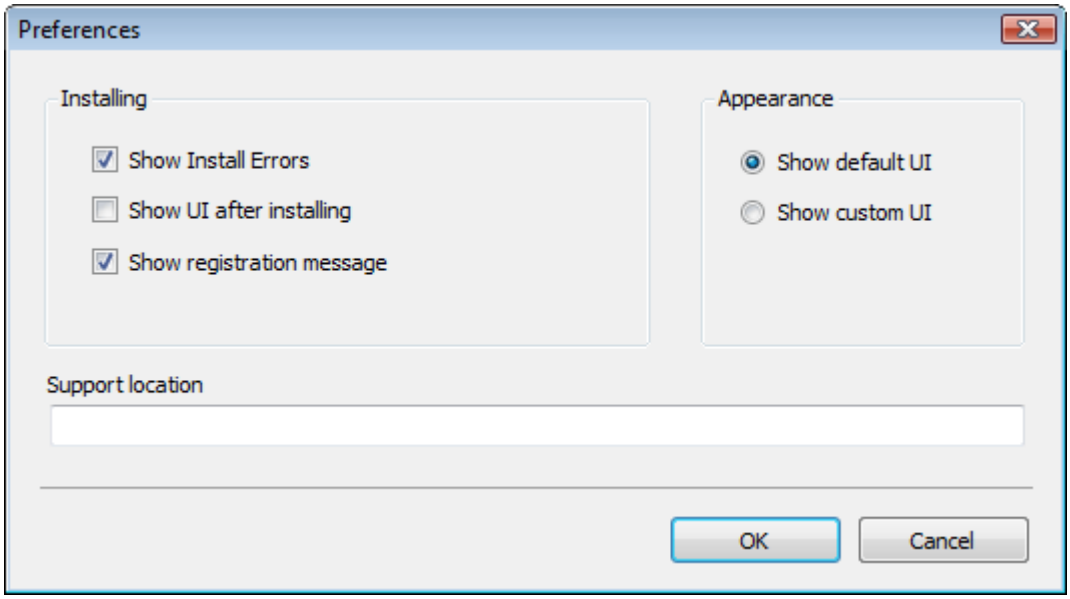
location for this version. In subsequent add-in versions, you can use the same or any other update location. For instance, you can use the same Provider URL in order to look for versions 1.0, 1.1, and 1.2 in one location and, when publishing version 1.3, specify another update location. Please note, that when the user updates the current version, he or she will get the most fresh add-in version existing in the location. That is, it is possible that the user updates from version 1.0 to version 1.3. The opposite is possible, too: this scenario requires the developer to publish v.1.3 and then re-publish v.1.0.

Step #3 - Signing the Manifests

Browse for the existing certificate file or click New to create a new one. Enter the password for the certificate (optional).

Step #4 - Preferences

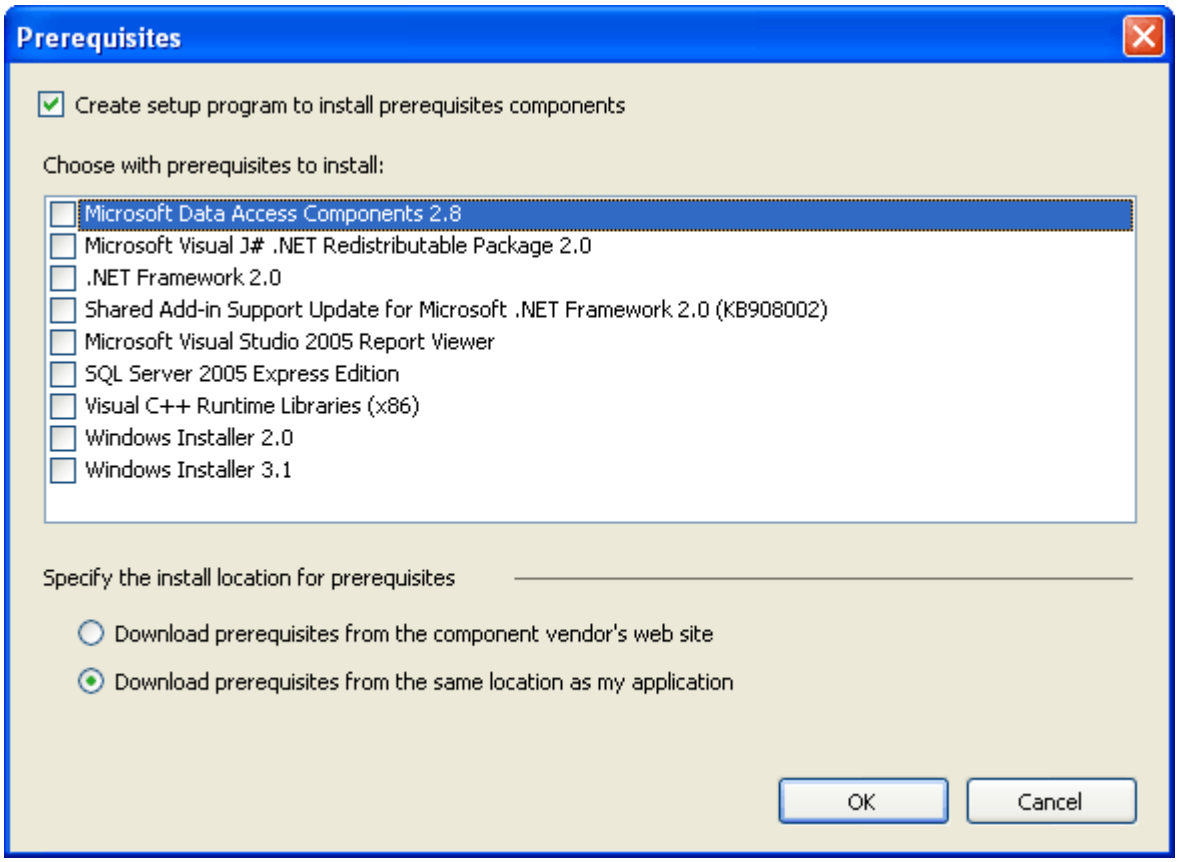
Click the Preferences button to open the following dialog window:



In this dialog, you specify if the ClickOnce module will get the OnShowClickOnceCustomUI event (it allows the add-in to show the custom UI), and provide the Support Location option for the Add Remove Programs dialog.

Step #5 - Prerequisites

When you click this button and select any prerequisites in the dialog, Add-in Express gathers the prerequisites you've chosen and creates a setup.exe to install them. Then you can upload the files to any appropriate location. When the user starts the setup.exe, it installs the prerequisites and invokes the ClickOnce API to install your add-in. Naturally, it may happen that some prerequisites can be installed by an administrator only. In this case, you may want to create a separate setup project that installs the prerequisites only and supply it to the administrator.



Step #6 – Publishing the Add-in

When you click on the *Publish* button, Add-in Express generates (updates) the manifests. Now you can copy files and folders of the *Publish* folder to a deployment location, say a web server. For testing purposes, you can just double-click the deployment manifest in Windows Explorer.

Manifest file names and locations

Deployment manifest – <SolutionFolder>/Publish/<projectname>.application

Application manifest - <SolutionFolder>/Publish/<ProjectVersion>/<ProjectName>.exe.manifest

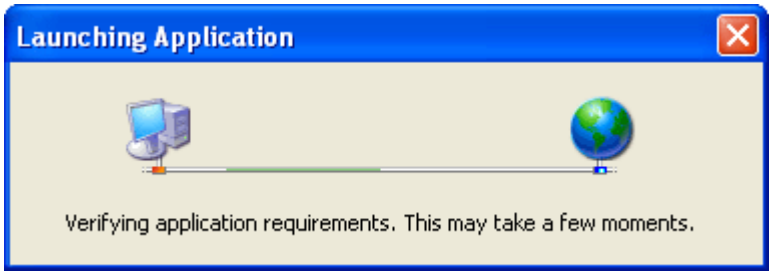
Step #7 – Publishing a New Add-in Version

In AssemblyInfo, change the version number and build the project. Click *Publish* and add the add-in files (*Populate* button). Fill in all the other fields. You can use the *Version* check box to switch to the data associated with any previous version.

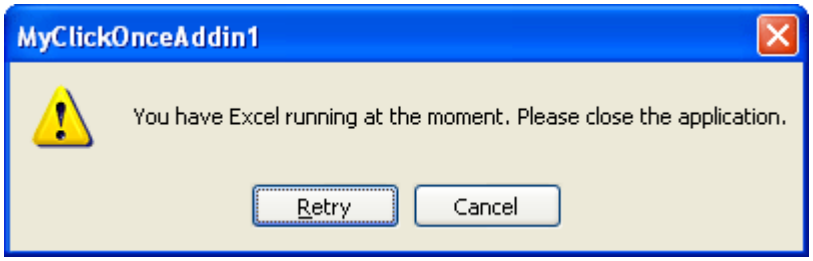
On the Target PC

Installing: User Perspective

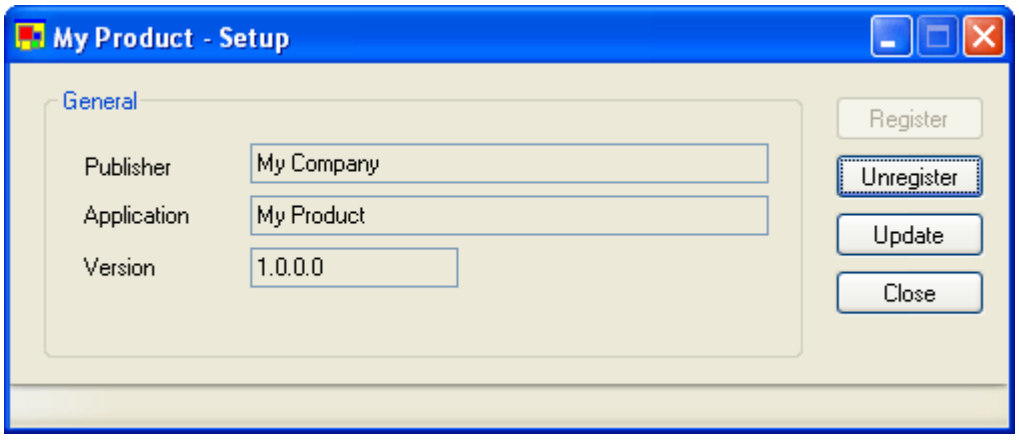
The user browses the deployment manifest (<projectname>.application) in either Internet Explorer or Windows Explorer and runs it. The following window is shown:



In accordance with the manifests, the ClickOnce loader will download the files to the ClickOnce Cache and run the launcher application. When run in this mode, it registers the add-in. If the host applications of the add-in are running at this moment, the user will be prompted to close them.



If the user clicks *Cancel*, the launcher will be installed, but the add-in will not be registered. However, in any appropriate moment, the user can click the launcher entry in the Start menu to run the launcher and register/unregister the add-in through the launcher GUI.



The current Add-in Express version relies on the name and location of the product entry in the Start Menu. Please, add this information to your user's guide.

Installing: Developer Perspective

If a ClickOnce module (ADXClickOnceModule) is added to your add-in project, you are able to handle all the actions applicable to add-ins: install, uninstall, register, unregister, update to a newer version, and revert to the previous version. For instance, you can easily imagine a form or wizard allowing the user to tune up the settings of your add-in. The ClickOnce module also allows you to show a custom GUI whenever the launcher application is required to show its GUI. If you don't process the corresponding event, the standard GUI of the Add-in Express ClickOnce application will be shown.

You can also make use of the ComRegisterFunction and ComUnRegisterFunction attributes in any assembly listed in the loader manifest (see `assemblyIdentity` tags). The methods marked with the ComRegisterFunction attribute will run when the add-in is registered. See MSDN for the description of the attributes.

Updating: User Perspective

The user can check for add-in updates in the launcher GUI (or in the GUI that you supply). To run it, the user clicks the entry in the Start Menu. If there is no update in the update location specified in the deployment manifest, an information message box is shown. If there is an update, the launcher requests the user to confirm his/her choice. If the answer is positive, the ClickOnce loader downloads new and updated files to the ClickOnce Cache, the launcher unregisters the current add-in version, restarts itself (this will run the launcher application supplied in the update files), and registers the add-in.

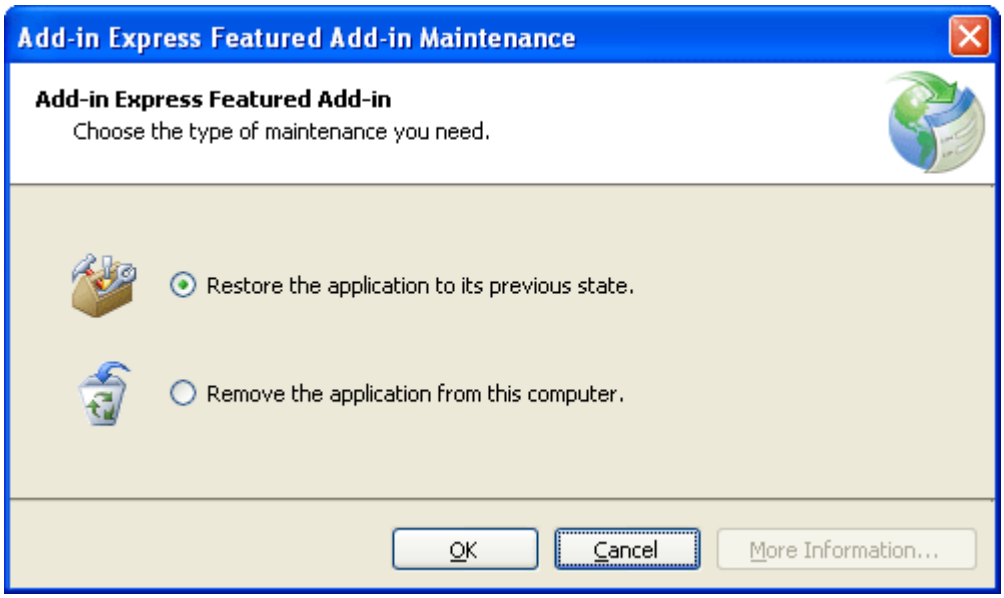
Updating: Developer Perspective

The add-in module provides you with the CheckForUpdates method. This method can result in one of the following ways:

- the add-in becomes updated;
- the ClickOnce module invokes the OnError event handler.

Uninstalling: User Perspective

To uninstall the add-in, the user goes to Add Remove Programs and clicks on the product name entry. This opens the following dialog.



- Restore the application to its previous state.

This option is disabled, if the add-in was never updated. If the user chooses this option, the Launcher is run, then it requires the user to close the host applications of your add-in, unregisters the add-in, requests ClickOnce API to start the Launcher application of the previous add-in version, and quits. After that, the Launcher application of the previous add-in version registers the add-in.

- Remove the application from this computer

This runs the Launcher that will require the user to close the host applications of your add-in. Then the Launcher unregisters the add-in and requests the ClickOnce API to delete both the add-in and the Launcher files.

Uninstalling: Developer Perspective

Handle the corresponding event of the ClickOnce module (ADXClickOnceModule) or use the ComUnRegisterFunction attribute to run your actions when the add-in is unregistered.

In the Web-based deployment scenario, the user can install an Office extension using Internet Explorer only. The ClickOnce article from Wikipedia states that Firefox allows ClickOnce-based installations too, but this was neither tested nor even verified.

Add-in Express Tips and Notes

You might have an impression that creating add-ins is a very simple task. Please don't get too enthusiastic. Sure, Add-in Express makes embedding your code into Office applications very simple, but you should write the applied code yourself, and we guess it would be something more intricate than a single call of MessageBox.

Development

Use the latest version of the loader

Since the code of the loader frequently changes, you must use its latest version. Whenever you install a new Add-in Express version, you need to unregister your add-in, copy adxloader.dll and adxloader64.dll located in {Add-in Express }\Redistributables to the Loader folder of you project; for XLL add-ins, you must also rename it to adxloader.{XLL add-in project name}.dll. After replacing the loader, you **must rebuild** (not just build) your project and register it. If everything was done correctly, you'll see the new loader version in adxloader.log (see Loader's Log).

Several Office Versions on the Machine

Although Microsoft allows installing multiple Office versions on a PC, it isn't recommended to do so. Below is a very long citation from an article by Andrew Whitechapel.

> First, the Office client apps are COM-based. Normal COM activation relies on the registry. COM registration is a "last one wins" model. That is, you can have multiple versions of a COM server, object, interface or type library on a machine at the same time. Also, all of these entities can be registered. However, multiple versions can (and usually do) use the same identifiers, so whichever version was registered last overwrites any previous one. Also, when it comes time to activate the object, only the last one registered will be activated. COM identity at runtime depends on an object's implementation of QueryInterface, but COM identity at the point of discovery depends on GUIDs. GUIDs are used because they provide a guaranteed (for all practical purposes) unique identifier (surprise).
>
> As soon as you put multiple versions of a COM server/object/interface/typelib onto the same machine, you introduce scope for variability. That is, although COM activation will ensure that the GUID-identified object gets used at the point of activation, you've set up the environment such that the object that this GUID identifies can change unpredictably over time – even short periods of time. This is one of the many reasons why it is very difficult to successfully develop solutions on a machine with multiple versions of Office – and one of the reasons we do not support this. But wait, how can this be? Surely a COM interface never versions? That's true, but, first, Office interfaces are not pure COM interfaces – they're automation interfaces, which are allowed to version (while retaining the same GUID). Second, the objects that implement the interfaces are obviously allowed to version, as are the typelibs that describe them.

Please read the rest of the article: Why is VS development not supported with multiple versions of Office?

Using threads

All object models provided by Office are not thread-safe. Using an object model from a thread other than the main one may produce unpredictable consequences. Once, we read Inspector.Count in a thread; after we stopped doing this, the users stopped complaining of a strange behavior of the *Down arrow* key when composing an e-mail.

When you need to use an object model in a thread, you can bypass this by using the SendMessage method and OnSendMessage event of the add-in module. One side of those members is described in Wait a Little. The other side is that the OnSendMessage event occurs in the main thread. That is, you can send a message from a thread and handle the message in the main thread.

Message Boxes When Debugging

Showing /closing a message box is accompanied by moving the focus off and back on to the host application window. When processing those actions, the host application generates a number of events (available for you through the corresponding object models). Therefore, showing a message box may mask the real flaw of events and you will just waste your time on fighting with windmills. We suggest using System.Diagnostics.Debug.WriteLine and the DebugView utility available on the Microsoft web site.

Releasing COM objects

When working with COM objects, remember these two rules:

- You **must never** release COM objects obtained through the parameters of events provided by Add-in Express.
- You **must always** release COM objects retrieved by you ("manually") from any COM object.

To understand why (and how) to release COM objects, consider the following code line:

C#:

```
Outlook.Explorer explorer = OutlookApp.ActiveExplorer();
```

VB.NET:

```
Dim explorer as Outlook.Explorer = OutlookApp.ActiveExplorer()
```

That code line creates three objects: a COM object corresponding to the active Outlook.Explorer and two .NET objects. The .NET objects are:

- A Runtime-callable wrapper (RCW) that references the COM object
- A .NET object that references the RCW. This .NET object is identified in your code as `explorer`.

When you set `explorer` to null (Nothing in VB.NET), the corresponding .NET object lives until the next run of the Garbage Collector (GC). Accordingly, the RCW lives, too. And this means the COM object isn't released. The further course of events depends on the COM object you created and the implementation of the COM server (it's Outlook in this example). Say, not releasing the COM object used in this example makes Outlook 2000 – 2002 hang in processes and produces a delay in Outlook 2003-2007 when you close Outlook; it isn't that hard to hang Outlook 2003 and 2007, though. In Outlook 2010, they introduced the feature called Fast Shutdown. With that feature enabled, your add-in developed for Outlook 2000-2007 doesn't have a chance to hang Outlook. But that feature comes at its price: your add-in isn't notified that Outlook is shutting down. Find more details about that feature and how to deal with it in Outlook 2010 Fast Shutdown feature published on Add-in Express blog.

To release the COM object above, you need to use the Marshal.ReleaseComObject method (System.Runtime.InteropServices namespace) as follows:

C#:

```
if (explorer != null) Marshal.ReleaseComObject(explorer);
```

VB.NET:

```
If explorer IsNot Nothing Then Marshal.ReleaseComObject(explorer)
```

An extensive review of typical problems related to releasing COM objects in Office add-ins is given in an article published on the Add-in Express technical blog – When to release COM objects in Office add-ins?.

Wait a Little

Some things aren't possible to do right at the moment; say, you can't close the inspector of an Outlook item in the Send event of that item. A widespread approach is to use a timer. Add-in Express provides a way to do this by using the <SendMessage> method and <OnSendMessage> event; when you call <SendMessage>, it posts the Windows message that you specified in the methods' parameters and the execution continues. When Windows delivers this message to an internal Add-in Express window, the <OnSendMessage> event is raised. Make sure that you filter incoming messages; there will be quite a lot of them.

The actual names of the <SendMessage> method and <OnSendMessage> event are listed below:

<SendMessage>

- ADXAddinModule.SendMessage
- ADXOlForm.ADXPostMessage
- ADXExcelTaskPane.ADXPostMessage
- ADXWordTaskPane.ADXPostMessage
- ADXPowerPointTaskPane.ADXPostMessage

<OnSendMessage>

- ADXAddinModule.OnSendMessage
- ADXOlForm.ADXPostMessageReceived
- ADXExcelTaskPane.ADXPostMessageReceived
- ADXWordTaskPane.ADXPostMessageReceived
- ADXPowerPointTaskPane.ADXPostMessageReceived

COM Add-ins

Getting help on COM objects, properties and methods

To get assistance with host applications' objects, their properties, and methods as well as help info, use the Object Browser. Go to the VBA environment (in the host application, choose menu *Tools* | *Macro* | *Visual Basic Editor* or just press *{Alt+F11}*), press *{F2}*, select the host application in the topmost combo and/or specify a search string in the search combo. Select a class /property /method and press *{F1}* to get the help topic that relates to the object.

An exception when registering /unregistering the add-in

When your add-in is registered and unregistered, Add-in Express creates an instance of the module. Because in this situation the module isn't loaded by the host application, you can't use any Office-related classes. If the code isn't prepared for this, it will break. If it breaks when you uninstall the add-in, you'll have to clean the registry either manually or using a registry cleaner program.

The same applies to class-level initializers; they are executed even before the module constructor is run.

To initialize your add-in, you need to use the AddinInitialize event of the module. It fires when Office loads the add-in. Note, however, that for Ribbon-enabled Office applications, the first event that the module fires is OnRibbonBeforeCreate.

The add-in doesn't work

See The add-in is not registered, An assembly required by your add-in cannot be loaded, An exception at add-in start-up, and Your add-in has fallen to Disabled Items.

The add-in is not registered

If LoadBehavior is 2, this may be an indication of an unhandled exception at add-in startup. Check Registry Keys.

An assembly required by your add-in cannot be loaded

Possible reasons are:

- the assembly is missing in the installer
- the user starting the host application doesn't have permissions for the folder where the add-in was installed; say, a per-machine add-in is installed to a user's Application Data folder and another user loads the add-in
- the PublicKeyToken of your add-in assembly doesn't correspond to the PublicKeyToken mentioned in the Add-in Express Loader Manifest. See below.

How to find the PublicKeyToken of the add-in

You can find it in the setup project, which must be already built. Click on your add-in primary output in the setup project and, in the Properties window, expand the KeyOutput property and see the PublicKeyToken property value.

An exception at add-in start-up

If an exception occurs in the constructor of the add-in module, or when module-level variables are initialized, Office will interrupt the loading sequence and set LoadBehavior of your add-in to 2. See Registry Keys.

Your add-in has fallen to Disabled Items

If your add-in fires exceptions at startup or causes the host application to crash, the host application (or the end-user) may block the add-in and move it to the Disabled Items list. To find the list, in Office 2000-2003, go to "Help", then "About". At the bottom of the About dialog, there is the *Disabled Items* button. Check it to see if the add-in is listed there (if so, select it and click *Enable*). In the Ribbon UI of Office 2007, you find that list on the Add-ins tab of the *Office Menu* | *{host application} Options* dialog. In the Ribbon UI of Office 2010, the Add-ins tab can be found in the *File* | *Options* dialog. After you get the *Disabled Items* dialog, you select the add-in and click *Enable*.

Delays at add-in start-up

If you use the WebViewPane layout of your Outlook forms, please check WebViewPane.

Try clearing the DLL cache - see Deploying – Shadow Copy.

Maybe you will be able to identify the source of the problem by turning off other COM add-ins and Smart Tags in the host application. If your host application is Excel, turn off all Excel add-ins, too. You can also try turning off your antivirus software.

Also, check http://office.microsoft.com/en-us/ork2003/HA011403081033.aspx.

Commands of the Add-in Module

The commands listed below are available in the context menu of the add-in module designer.

- Add CommandBar – allows creating a custom toolbar or modifying an existing (or built-in) toolbar of the host application (see Command Bar UI)
- Add Main Menu – allows modifying the main menu of the host application (see Command Bar UI)
- Add Context Menu – allows modifying context menus of the host application (see Command Bar UI)
- Add Explorer CommandBar – allows creating a custom toolbar or modifying an existing (or built-in) toolbar in Outlook Explorer windows (see Command Bar UI)
- Add Explorer Main Menu – allows modifying the main menu in Outlook Explorer windows (see Command Bar UI)
- Add Inspector CommandBar – allows creating a custom toolbar or modifying an existing (or built-in) toolbar in Outlook Inspector windows (see Command Bar UI)
- Add Inspector Main Menu – allows modifying the main menu in Outlook Inspector windows (see Command Bar UI)
- Add Built-in Control Connector – allows intercepting the action of a built-in control of the host application (see Connecting to Existing CommandBar Controls)
- Add Keyboard Shortcut – allows creating and intercepting application-level keyboard shortcuts (see Intercepting Keyboard Shortcuts)
- Add Outlook Bar Shortcut Manager – allows adding Outlook Bar shortcuts and shortcut groups (see Outlook Bar Shortcut Manager)
- Add Ribbon Tab – allows creating a custom Ribbon tab or modifying an existing (or built-in) Ribbon tab of the host application (see Ribbon UI)
- Add Ribbon Command – allows re-purposing Ribbon controls (see Ribbon UI).
- Add Ribbon Quick Access Toolbar – allows customizing the Ribbon Quick Access Toolbar (see Ribbon UI)
- Add Ribbon Office Menu – allows customizing the Ribbon Office Menu (see Ribbon UI)
- Add Excel Task Panes Manager – see Excel Task Panes
- Add Outlook Forms Manager – see Advanced Outlook Regions
- Add Events – allows accessing application-level events of the host application (see Application-level Events)
- Host configuration – see Conflicts with Office extensions developed in .NET Framework 1.1

What is ProgID?

ProgID = Program Identifier. This is a textual name representing a server object. It consists of the project name and the class name, like MyServer.MyClass.

You find it in ProgIDAttribute of an add-in module. For instance:

```
...
'Add-in Express Add-in Module
<GuidAttribute("43F48D82-7C6F-4705-96BB-03859E881E2C"), _
       ProgIdAttribute("MyAddin1.AddinModule")> _
Public Class AddinModule
    Inherits AddinExpress.MSO.ADXAddinModule
...
```

We found the definition of ProgID in The COM / DCOM Glossary. On that page, you can find other COM-related terms and their definitions.

FolderPath Property Is Missing in Outlook 2000 and XP

The function returns the same value as the MAPIFolder.FolderPath property available in Outlook 2003 and higher.

```
   Private Function GetFolderPath(ByVal folder As Outlook.MAPIFolder) _
      As String

      Dim path As String = ""
      Dim toBeReleased As Boolean = False
      Dim tempObj As Object = Nothing

      While folder IsNot Nothing
         path = "\" + folder.Name + path
         Try
            tempObj = folder.Parent
         Catch
            'permissions aren't set
            tempObj = Nothing
         Finally
            If toBeReleased Then
               Marshal.ReleaseComObject(folder)
            Else
               'the caller will release the folder passed
               toBeReleased = True
```

```
            End If
            folder = Nothing
         End Try

         'The parent of a root folder is of the Outlook.Namespace type
         If TypeOf tempObj Is Outlook.MAPIFolder Then
            folder = CType(tempObj, Outlook.MAPIFolder)
         End If
      End While

      If tempObj IsNot Nothing Then Marshal.ReleaseComObject(tempObj)
      If path <> "" Then path = Mid$(path, 2)
      Return path
   End Function
```

Word add-ins, command bars, and normal.dot

Word saves changes in the UI to normal.dot: move a toolbar to some other location and its position will be saved to normal.dot when Word quits. The same applies to add-ins: their command bars are saved to this file. See some typical support cases related to Word add-ins and normal.dot below.

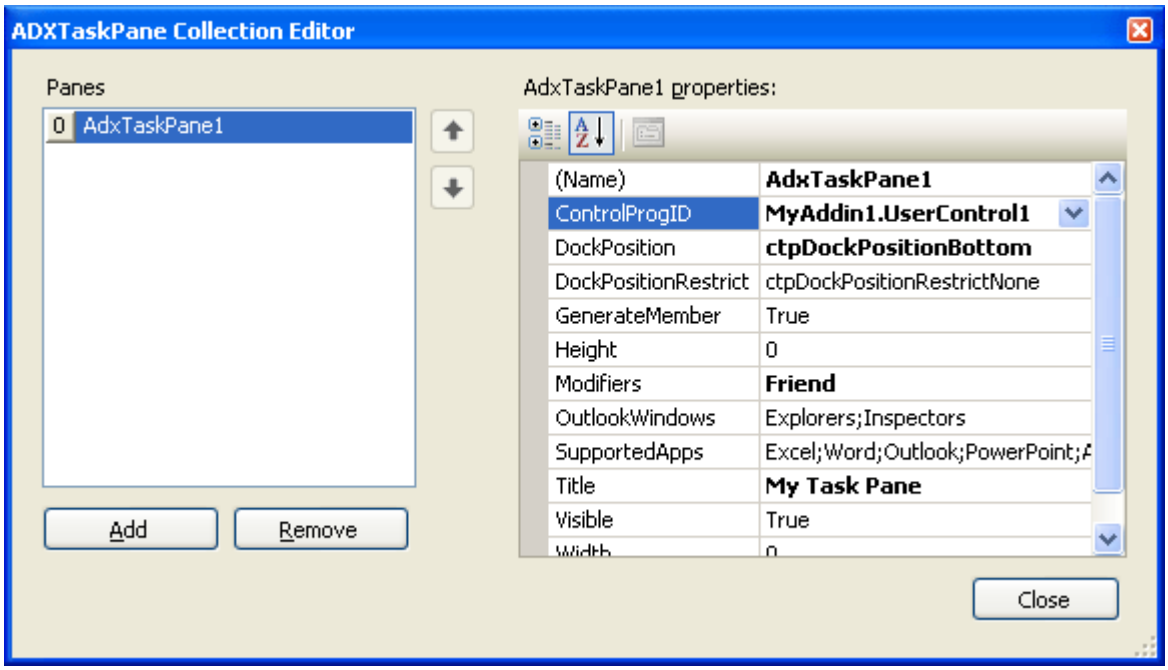
- For reasons of their own, some organizations use read-only normal.dots. In this case, installing the add-in raises a warning, when Word tries to save normal.dot and finds that it is read-only.
- The user can set the *Prompt to Save Normal Template* flag located on the *Save* tab in the *Tools* | *Options* menu and in this way decide whether to save normal.dot or not. This may lead to a mess: some command bars and controls are saved while others are not.
- Other companies store lots of things in their normal.dot files making them too big in size; saving such files requires extra time.
- We have had scenarios in which normal.dot is moved or deleted after the add-in is installed; naturally, command bars disappear as well.

You may think that using temporary command bars in these cases is a way out, but this may not be your case: see How Command Bars and Their Controls Are Created and Removed?

We know the only workaround: don't use normal.dot in a way, which wasn't designed by Microsoft. Normal.dot is a per-user thing. Don't deprive the user of the ability to change its UI. Move all excessive things to other templates. Always insist on clearing the *Prompt to Save Normal Template* flag. If it is possible, of course...

Custom Task Panes (Office 2007+)

To add a new task pane, you add a UserControl to your project and populate it with controls. Then you add an item to the TaskPanes collection of the add-in module and specify its properties:



- Caption – the caption of your task pane (required!)
- Height, Width – the height and width of your task pane (applies to horizontal and vertical task panes, correspondingly)
- DockPosition – you can dock your task pane to the left, top, right, or bottom edges of the host application window
- ControlProgID – the UserControl just added

In Add-in Express, you work with the task pane component and task pane instances. The TaskPanes collection of the add-in module contains task pane components of the AddinExpress.MSO.ADXTaskPane type. When you set, say, the height or dock position of the component, these properties apply to every task pane instance that the host application shows. To modify a property of a task pane instance, you should get the instance itself. This can be done through the Item property of the component (in C#, this property is the indexer for the ADXTaskPane class); the property accepts a window object (such as Outlook.Explorer, Outlook.Inspector, Word.Window, etc) as a parameter and returns an AddinExpress.MSO.ADXTaskPane.ADXCustomTaskPaneInstance representing a task pane instance. For example, the method below finds the currently active instance of the task pane in Outlook 2007 and refreshes it. For the task pane to be refreshed in a consistent manner, this method should be called in appropriate event handlers.

```
Private Sub RefreshTaskPane(ByVal ExplorerOrInspector As Object)
   If Me.HostVersion.Substring(0, 4) = "12.0" Then
      Dim TaskPaneInstance As _
         AddinExpress.MSO.ADXTaskPane.ADXCustomTaskPaneInstance = _
            AdxTaskPane1.Item(ExplorerOrInspector)
      If Not TaskPaneInstance Is Nothing _
```

```
        And TaskPaneInstance.Visible Then
        Dim uc As UserControl1 = TaskPaneInstance.Control
        If Not uc Is Nothing Then _
          uc.InfoString = GetSubject(ExplorerOrInspector)
      End If
   End If
End Sub
```

The InfoString property just gets or sets the text of the Label located on the UserControl1. The GetSubject method is shown below.

```
   Private Function GetSubject(ByVal ExplorerOrInspector As Object) _
      As String
      Dim mailItem As Outlook.MailItem = Nothing
      Dim selection As Outlook.Selection = Nothing

      If TypeOf ExplorerOrInspector Is Outlook.Explorer Then
         Try
            selection = CType(ExplorerOrInspector, _
               Outlook.Explorer).Selection
            mailItem = selection.Item(1)
         Catch
         Finally
            If Not selection Is Nothing Then _
               Marshal.ReleaseComObject(selection)
         End Try
      ElseIf TypeOf ExplorerOrInspector Is Outlook.Inspector Then
         Try
            mailItem = CType(ExplorerOrInspector, _
               Outlook.Inspector).CurrentItem
         Catch
         End Try
      End If

      If mailItem Is Nothing Then
         Return ""
      Else
         Dim subject As String = "The subject is: " + mailItem.Subject
         Marshal.ReleaseComObject(mailItem)
         Return subject
      End If
   End Function
```

The code of the GetSubject method emphasizes the following:

- The ExplorerOrInspector parameter was originally obtained through parameters of Add-in Express event handlers. That is why we do not release it (see Releasing COM objects).

- The `selection` and `mailItem` COM objects were created "manually" so they must be released.
- All Outlook versions fire an exception when you try to obtain the Selection object for a top-level folder, such as *Personal Folders*.

Below is another sample that demonstrates how the same things can be done in Excel or Word.

```
Imports AddinExpress.MSO
...
   Private Sub RefreshTaskPane()
      If Version = "12.0" Then
         Dim Window As Object = Me.HostApplication.ActiveWindow
         If Not Window Is Nothing Then
            RefreshTaskPane(AdxTaskPane1.Item(Window))
            Marshal.ReleaseComObject(Window)
         End If
      End If
   End Sub

   Private Sub RefreshTaskPane(ByVal TaskPaneInstance As _
      ADXTaskPane.ADXCustomTaskPaneInstance)
      If Not TaskPaneInstance Is Nothing Then
         Dim uc As UserControl1 = TaskPaneInstance.Control
         If uc IsNot Nothing And TaskPaneInstance.Window IsNot Nothing Then
            uc.InfoString = GetInfoString(TaskPaneInstance.Window)
         End If
      End If
   End Sub
```

The InfoString property mentioned above just updates the text of the label located on the UserControl. Please pay attention to Releasing COM objects in this code.

Custom Actions When Your COM Add-in Is Uninstalled

When the add-in is being unregistered, the BeforeUninstallControls and AfterUninstallControls events occur. You can use them for, say, removing “hanging” command bars from Word or restoring any other state that should be restored when your add-in is uninstalled.

XP Styles in Your Forms

Just call System.Windows.Forms.Application.EnableVisualStyles() in your add-in module, say in the AddinInitialize event.

Command Bars and Controls

Command Bar Terminology

In this document, on our site, and in all our texts we use the terminology suggested by Microsoft for all toolbars, their controls, and for all interfaces of the Office type library. For example:

- Command bar is a toolbar, a menu bar, or a context menu.
- Command bar control is one of the following: a button (menu item), edit box, combo box, or pop-up.
- Pop-up can stand for a pop-up menu, a pop-up button on a command bar or a submenu on a menu bar.

According to help files, a pop-up control is a built-in or custom control on a menu bar or toolbar that displays a menu when it's clicked, or a built-in or custom menu item on a menu, submenu, or shortcut menu that displays a submenu when the pointer is positioned over it.

Pop-up button samples are View and View | Toolbars in the main menu and Draw in the Drawing toolbar in Word or Excel.

ControlTag vs. Tag Property

Add-in Express identifies all its controls (command bar controls) using the ControlTag property which is mapped to the the Tag property of the CommandBarControl interface. The value of this property is generated automatically and you do not need to change it. For your own needs, use the Tag property of the command bar control instead.

Pop-ups

According to the Microsoft's terminology, the term "pop-up" can be used for several controls: pop-up menu, pop-up button, and submenu. With Add-in Express, you can create a pop-up as using the Controls property of a command bar and then add any control to the pop-up via the Controls property of the pop-up.

However, pop-ups have an annoying feature: if an edit box or a combo box is added to a pop-up, their events are fired very oddly. Please don't regard this bug as that of Add-in Express.

Built-in Controls and Command Bars

You can connect an ADXCommandBar instance to any built-in command bar. For example, you can add your own controls to the *"Standard"* command bar or remove some controls from it. To do this just add to the add-in module a new ADXCommandBar instance and specify the name of the built-in command bar you need via the CommandBarName property.

You can add any built-in control to your command bar. To do this, just add an ADXCommandBarControl instance to the ADXCommandBar.Controls collection and specify the ID of the required built-in control in the ADXCommandBarControl.Id property. To find out the built-in control IDs, use the free Built-in Controls Scanner utility (http://www.add-in-express.com/downloads/controls-scanner.php).

CommandBar.SupportedApps

Use this property to specify if the command bar will appear in some or all host applications supported by the add-in. Unregister your add-in before you change the value of this property.

Outlook CommandBar Visibility Rules

Add-in Express displays the Explorer command bar for every folder, which name **AND** type correspond to the values of FolderName, FolderNames, and ItemTypes properties. For the Inspector toolbar, the same rule applies to the folder in which an Outlook item is opened or created.

COM Add-ins for Outlook – Template Characters in FolderName

Notwithstanding the fact that the default value of the FolderName property is '*' (asterisk), which means "every folder", the current version doesn't support template characters in the FolderName(s) property value. Moreover, this is the only use of the asterisk recognizable in the current version.

Removing Custom Command Bars and Controls

Add-in Express removes custom command bars and controls while the add-in is uninstalled. However, this doesn't apply to Outlook and Access add-ins. You should set the Temporary property of custom command bars (and controls) to true to notify the host application that it can remove them itself. If you need to remove a toolbar or button manually, use the *Tools* | *Customize* dialog. See also Custom Actions When Your COM Add-in Is Uninstalled.

CommandBar.Position = adxMsoBarPopup

This option allows displaying the command bar as a popup (context) menu. In the appropriate event handler, you write the following code:

```
AdxOlExplorerCommandBar1.CommandBarObj.GetType.InvokeMember("ShowPopup", _
    Reflection.BindingFlags.InvokeMethod, Nothing, _
    AdxOlExplorerCommandBar1.CommandBarObj, Nothing)
```

The same applies to other command bar types.

Built-in and Custom Command Bars in Ribbon-enabled Office Applications

Do you know that all usual command bars that we used in earlier Office versions are still alive in Office 2007-2010 applications? For instance, our free Built-in Controls Scanner (http://www.add-in-express.com/downloads/controls-scanner.php) reports that Outlook 2007 e-mail inspector still has the *Standard* toolbar with the *Send* button on it. This may be useful if the functionality of your add-in takes into account the enabled/disabled state of this or that toolbar button.

As to custom toolbars, you can use set the UseForRibbon property of the corresponding component to true (the default value is false). This will result in your command bar controls showing up on the *Add-ins* tab along with command bar controls from other add-ins.

Transparent Icon on a CommandBarButton

It looks like the ImageList has a bug: when you add images and then set the TransparentColor property, it corrupts the images in some way. Follow the steps below (at design-time) to get your images transparent:

- Make sure the ImageList doesn't contain any images;
- Set its TransparentColor property to Transparent;
- Add images to the ImageList;
- Choose an image in the Image property of your command bar button;
- Specify the transparent color in the ImageTransparentColor property of the command bar button;
- Rebuild the project.

Navigating Up and Down the Command Bar System

It is easy to navigate down the command bar system: the host application supplies you with the Office.CommandBars interface that provides the Controls property returning a collection of the Office.CommandBarControls type. Office.CommandBarPopup provides the Controls property, too.

When navigating up the command bar system, you use the Parent property of the current object. For a command bar control (see Command Bar Terminology), this property returns Office.CommandBar. Note that the same applies to controls on a pop-up; command bars returned in this way aren't listed anywhere else in the command bar system. The parent for an Office.CommandBar is the host application. The parent for an Outlook command bar is either Outlook.Inspector or Outlook.Explorer.

Hiding and Showing Outlook Command Bars

ADXOlExplorerCommandBar and ADXOlInspectorCommandBar implement context-sensitive command bars; when the current folder correspond to the components' settings, the corresponding command bar is shown. To "manually" hide or show an inspector comand bar, you handle the InspectorActivate event of the Outlook

Application Events component (ADXOutlookAppEvents) and set the Visible property of the ADXOlInspectorCommandBar to an appropriate value.

Explorer command bars are handled in the ExplorerFolderSwitch event (see ADXOutlookAppEvents). One thing to remember: you need to set ADXOlExplorerCommandBar.Enabled to true before you change ADXOlExplorerCommandBar.Visible to true.

To hide an Outlook command bar "forever", you set the FolderName property of the corresponding command bar component so that it never matches any Outlook folder name.

Debugging and Deploying

Conflicts with Office extensions developed in .NET Framework 1.1

In general case, two Office extensions based on .NET Framework 1.1 and 2.0 (or higher), will not work together. That occurs because of three facts:

- Before they introduced .NET Framework 4.0, two .NET Framework versions could not be loaded in the same Windows process. If there were two Office extensions written in .NET Framework 1.1 and 2.0 (3.0 and 3.5 are just extensions of 2.0), the first extension loads its .NET Framework version and the second extension is obliged to use the same .NET Framework. Now, with NET Framework 4.0, an add-in based on .NET Framework 1.1 will prevent add-ins based on .NET Framework 2.0 from loading and vice-versa.
- There are Breaking Changes between .NET Framework 1.1 and 2.0.
- You can't influence the order in which Office extensions are loaded; however, you can choose all Office extensions to use the same.NET Framework version – see below.

We suggest checking the environments in which your would-be add-in will work. First off, you need to look for a .config file(s) for the host application of your add-in. The examples of configuration file names are outlook.exe.config and excel.exe.config. If such a file exists, it is located in the Office folder; say, for Office 2003, the folder is C:\Program Files\Microsoft Office\OFFICE11. Open such a file in any text editor and see if a .NET Framework version is specified; if it is specified, then all extensions loaded by that host application(s) use the specified .NET Framework version.

If you spotted an extension using different .NET Framework version, then, in the worst case, you will need either to turn it off, or use the same .NET Framework version in your project.

However, all of the things above will not help because the end-user may install an add-in based on the other .NET Framework version after you install your add-in, smart tag, etc. So, a simple conclusion is **never use Visual Studio 2003 to develop Office extensions**.

Always check the log file (see Loader's Log) for the CLR version that is used for your add-on. If you run into a situation of two conflicting add-ins, you can try to create a .config file pointing to a .NET Framework version of

your choice and copy that file to the Office folder on the target machine. To create such a file on your PC, you use the *Host Configuration* command of the COM add-in module (create an empty COM add-in project, if required). This command [creates and] changes the configuration file for the host application of your COM add-in. When you are done, don't forget to use the *Host Configuration* command again to restore the state. Other ways are to turn the conflicting add-in off, or use the same .NET Framework version in your project.

For All Users or For the Current User?

COM add-ins and RTD servers can be registered either for the current user (the user the permissions of which are used to run the installer) or for all users on the machine. That's why the corresponding module types provide the RegisterForAllUsers property. Registering for all users means writing to HKLM and therefore the user registering a per-machine extension must have administrative permissions. Accordingly, RegisterForAllUsers = Flase means writing to HKCU (=for the current user) and therefore such an Office extension can be registered by a standard user.

Add-ins deployed via ClickOnce can write to HKCU only.

The setup project wizard analyzes RegisterForAllUsers and creates a setup project that is ready to install the files mentioned in Files to Deploy to the following default locations:

RegisterForAllUsers = True	RegisterForAllUsers = Flase
[ProgramFilesFolder][Manufacturer]\[ProductName]	[AppDataFolder][Manufacturer]\[ProductName]

All other Office extensions can be installed for the current user only.

Updating on the fly

It isn't possible to update an Office extension on the fly. That's because Office loads the extension and to unload it and free its resources, you have to close the host application(s) of the extension.

User Account Control (UAC) on Vista, Windows 7 and Windows Server 2008

The User Account Control (UAC) should be turned on Vista; it should be set to the default level on windows 2008 Server and Windows 7. This is necessary when you install a COM add-in for all users on the PC, that is, when the RegisterForAllUsers property of the add-in module is true. Note that when UAC is off, a per-user add-in (RegisterForAllUsers = false) installed by an administrator will not work. This is restriction of systems with UAC.

Deploying Word add-ins

If your add-in delivers custom or customizes built-in command bars in any Word version, it isn't recommended setting the RegisterForAllUsers property of the add-in module to True. Since Word saves custom command

bars and controls to normal.dot, every user has its own copy of command bars saved to their normal.dot. And when the administrator uninstalls the add-in, the command bars will be removed for the administrator only.

See also Word add-ins, command bars, and normal.dot and How Command Bars and Their Controls Are Created and Removed?

InstallAllUsers Property of the Setup Project

The InstallAllUsers property sets the default state of the "Install {setup project title} for yourself, or for anyone who uses this computer" group of option buttons (they are "Everyone" and "Just me") in the installer. This group, however, is hidden by the executable mentioned in the PostBuildEvent property of the setup project generated by Add-in Express. This is done because to install your Office extension for all users on the machine you need to use the RegisterForAllUsers property of the corresponding module (add-in module, RTD module, etc). To find that property, open the module's designer (see Add-in Express Basics), click its surface and see the *Properties* window.

See also Deploying Word add-ins.

COM Add-ins Dialog

In Office 2010 you click *File Tab* | *Options* and, on the Add-ins tab, choose COM Add-ins in the Manage dropdown and click Go.

In version 2007 of Word, Excel, PowerPoint and Access you click the Office Menu button, then click {Office application} options and choose the Add-ins tab. Now choose COM Add-ins in the Manage dropdown and click Go.

In all other Office applications, you need to add the COM Add-ins command to a toolbar or menu of your choice. To do so, follow the steps below:

- Open the host application (Outlook, Excel, Word, etc)
- On the Tools menu, click Customize.
- Click the Commands tab.
- In the Categories list, click the Tools category.
- In the Commands list, click COM Add-Ins and drag it to a toolbar or menu of your choice.

In Office 2000-2003, the COM Add-ins dialog shows only add-ins registered in HKCU. In Office 2007-2010, HKLM-registered add-ins are shown too. See also Registry Keys.

Deploying – Shadow Copy

The Add-in Express loader uses the *ShadowCopy*-related properties and methods of the AppDomain class. When you run your add-on, the host application loads the Add-in Express loader DLL referenced in the registry. The loader does the following:

- It finds your add-on DLLs in the DLL Cache. If there are no add-in DLLs in the cache, it copies all assemblies to the cache (including dependencies). The cache folder is located in C:\Documents and Settings\<user name>\Local Settings\Application Data\assembly\dl<number>. If all add-in DLLs (including dependencies) already exist in the cache, it compares their versions. If the versions are not the same, it copies new DLLs to the cache.
- It loads the add-on DLLs from the cache.

You can see how the add-on versioning influences the add-in loading.

This approach (it is built into .NET, as you can see) allows replacing add-in DLLs when the add-in is loaded. The disadvantage is numerous files located in the cache. As far as we know, Microsoft doesn't provide a solution for this problem. You may think you can remove these files in an add-in's uninstall custom action. However, this will remove the files from the current profile only.

Deploying – "Everyone" Option in a COM Add-in MSI package

The *Everyone* option of the MSI installer doesn't have any effect on the Add-in Express based COM add-ins and RTD servers. See also InstallAllUsers Property of the Setup Project.

Deploying Office Extensions

Make sure that Windows and Office have all updates installed: Microsoft closes their slips and blunders with service packs and other updates. Keep an eye on Visual Studio updates, too.

If you deploy a per-user Office extension such as a per-user COM add-in or RTD server having RegisterForAllUsers= False in their modules as well as an Excel UDF or smart tag) **and** no pre-requisites requiring administrative permissions are used, a standard user can install the Office extension by running the .MSI file. If you deploy a per-machine Office extension (a COM add-in or RTD server having RegisterForAllUsers= True in their modules) or if prerequisites requiring administrative permissions are used, an administrator must run the bootstrapper (setup.exe).

Note that if a standard user runs setup.exe on Vista, Windows 7 or Windows 2008 Server with UAC turned on, the elevation dialog may pop up and this may end with installing the add-in to the admin profile. In such a case, the add-in will not be available for the standard user. But on the other hand, this installs pre-requisites and makes possible installing the Office extension for the standard user by running the .MSI file.

ClickOnce Cache

The cache location is visible in the COM Add-ins Dialog. It may have the following look:

```
C:\Documents and Settings\user\Local Settings\Apps\2.0\NCPNO3QK.0KJ\ONNRMXC3.ALM\add-..d-in_5c073faf40955414_0001.0000_2a2d23ab74b720da
```

Currently, we don't know if there is a decent way to clear the cache.

ClickOnce Deployment

Make sure that your IIS is allowed to process .application files. For instance, on a PC of ours, we had to edit the urlscan.ini file created by *UrlScan* (see http://support.microsoft.com/kb/307608). The only change was adding the .application extension to the *AllowExtensions* list. The full file name is C:\WINDOWS\system32\inetsrv\urlscan\urlscan.ini.

Customizing Dialogs When Updating the Add-in via ClickOnce

ClickOnce doesn't provide any opportunity to customize or hide dialogs and messages shown while the user updates your add-in.

Excel UDFs

My Excel UDF Doesn't Work

You start finding the cause from Use the latest version of the loader**Error! Reference source not found.**.

If your UDF isn't shown in the Add-in Manager dialog, then it isn't registered – see Locating Excel UDF Add-ins in the Registry.

Then you need to check the log file (see Loader's Log) for errors. If there are no errors but both .NET Framework 1.1 and 2.0 are mentioned in the log, read Conflicts with Office extensions developed in .NET Framework 1.1. Another typical problem is described in XLL and Shared Add-in Support Update.

My XLL Add-in Doesn't Show Descriptions

When you enter a formula in the *Formula Bar*, neither function descriptions nor descriptions of function parameters are shown. Debugging this problem shows that Excel just doesn't call any methods responsible for providing that info.

Also, we've found a non-described restriction in XLLs: the total length of a string containing all parameter names of a given function divided by a separator character minus one cannot be greater than 255. The same restriction applies to parameter descriptions. If any of such strings exceed 255 characters, many strange things

occur with the descriptions in the Excel UI. Below there are two useful functions that help checking parameter names and descriptions; add those functions to the XLLContainer class of your XLL module and invoke them in an Excel formula.

```
Imports AddinExpress.MSO
...
Public Shared Function GetParameterNames(ByVal fName As String)
   Dim names As String = "not found"
   For Each comp As Object In _Module.components.Components
      If TypeOf comp Is ADXExcelFunctionDescriptor Then
         Dim func As ADXExcelFunctionDescriptor = comp
         If func.FunctionName.ToLower = fName.ToLower Then
            names = ""
            For Each desc As ADXExcelParameterDescriptor In _
               func.ParameterDescriptors
               names += IIf(desc.ParameterName Is Nothing, "", _
                  desc.ParameterName) + ";"
            Next
            names = names.Substring(0, names.Length - 1)
            names = names.Length.ToString() + "=" + names
         End If
      End If
   Next
   Return names
End Function

Public Shared Function GetParameterDescriptions(ByVal fName As String)
   Dim descriptions As String = "not found"
   For Each comp As Object In _Module.components.Components
      If TypeOf comp Is ADXExcelFunctionDescriptor Then
         Dim func As ADXExcelFunctionDescriptor = comp
         If func.FunctionName.ToLower = fName.ToLower Then
            descriptions = ""
            For Each desc As ADXExcelParameterDescriptor In _
               func.ParameterDescriptors
               descriptions += IIf(desc.Description Is Nothing, "", _
                  desc.Description) + ";"
            Next
            descriptions = descriptions.Substring(0, descriptions.Length - 1)
            descriptions = descriptions.Length.ToString() + "=" + descriptions
         End If
      End If
   Next
   Return descriptions
End Function
```

Can an Excel UDF Return an Object of the Excel Object Model?

A UDF may return a value of any object type, of course. However, the UDF is always called in a certain Excel context and this makes impossible some things that are possible in other contexts: say, when called in a UDF returning an Excel.Hyperlink, the Hyperlinks.Add method inserts a hyperlink displaying an error value (#Value!) and working properly in all other respects. The same code works without any problems when called from a button created by a COM add-in.

Can an Excel UDF Change Multiple Cells?

Usually a UDF returns a single value. When called from an array formula, the UDF can return a properly dimensioned array (see Returning Values When Your Excel UDF Is Called From an Array Formula). Changing arbitrary cells from a UDF may crash or hang Excel.

Using the Excel Object Model in an XLL

At http://support.microsoft.com/kb/301443, they say:

> A function that is defined in an XLL can be called under three circumstances:
>
> 1. During the recalculation of a workbook
> 2. As the result of Excel's Function Wizard being called on to help with the XLL function
> 3. As the result of a VBA macro calling Excel's **Application.Run** Automation method
>
> Under the first two circumstances, Excel's Object Model does not expect, and is not prepared for, incoming Automation calls. Consequently, unexpected results or crashes may occur.

So, you must be prepared for the fact that some calls to the Excel Object model from your UDF may crash or hang Excel.

Determining What Cell / Worksheet / Workbook Your UDF Is Called From

In your Excel Automation add-in, you cast the ADXExcelAddinModule.HostApplication property to Excel.Application and get `ExcelApp.Caller` in VB or call `ExcelApp.get_Caller(Type.Missing)` in C#. That method **typically** returns an Excel.Range containing the cell(s) the UDF is called from (see the Excel VBA Help Reference on Application.Caller).

In your XLL add-in, you use the ADXXLLModule.CallWorksheetFunction method. The ADXExcelRef returned by that method allows determining the index (indices) of the cell(s) on the worksheet the UDF is called from. You can also call the ADXExcelRef.ConvertToA1Style (or ConvertToR1C1Style) method and get a string representing the caller's address, which is convertible to an Excel.Range by passing it to the `_Module.ExcelApp.Range` method (in C#, the second parameter of the Range method is `Type.Missing`).

The `_Module` (`Module` in C#) above is an automatically generated property of the XLLContainer class. The `ExcelApp` above is an automatically generated property of the XLLModule class.

Determining if Your UDF Is Called from the Insert Formula Dialog

The Insert Formula Dialog starts a one-step wizard that calls your UDF in order to provide the user with the description of the UDF parameters (XLL only), the current return value as well as with an entry point to the help reference for your UDF. Say in your XLL, you can use the AddinExpress.MSO.ADXXLLModule.IsInFunctionWizard property to return a string describing the actual return value.

In an Excel Automation add-in, you can use the Win API to find if the wizard window is shown. You can also try another approach suggested by a customer (thank you, Chris!):

```
private bool InFunctionWizard
{
    get
    {
        return (ExcelApp.CommandBars["Standard"].Controls[1].Enabled == false);
    }
}
```

Returning an Error Value from an Excel UDF

In Excel Automation add-ins, you use AddinExpress.MSO.ADXExcelError. In XLL add-ins, see AddinExpress.MSO.ADXxlCVError.

Returning Values When Your Excel UDF Is Called From an Array Formula

Just return a properly dimensioned array of a proper type. You can find the array dimensions from the range the UDF is called from – see Determining What Cell / Worksheet / Workbook Your UDF Is Called From. Here are two useful XLL samples.

```
// - select 3 consequent cells in a row,
// - enter "=GetRow()"
// - press Ctrl+Shift+Enter
public static object[] GetRow()
{
    object[] retVal = new object[3] { 1, 2, 3 };
    return retVal;
}

// - select 3 consequent cells in a column,
```

```
// - enter "=GetColumn()"
// - press Ctrl+Shift+Enter
public static object[,] GetColumn()
{
    object[,] retVal = new object[3, 1] { { 0 }, { 1 }, { 2 } };
    return retVal;
}
```

XLL and Shared Add-in Support Update

If you develop an XLL in VS 2005 or VS 2008, you might need to add Shared Add-in Support Update (KB908002) to prerequisites of your setup project. While the article clearly states that it relates to VS 2005 only, it does apply to development of an XLL in VS 2008. To add the update to the Prerequisites dialog of VS 2008, install the update and copy the following folder

C:\Program Files\Microsoft Visual Studio 8\SDK\v2.0\BootStrapper\Packages\KB908002\

to the following one:

C:\Program Files\Microsoft SDKs\Windows\v6.0A\Bootstrapper\Packages

This adds Shared Add-in Support Update to the list of pre-requisites in VS 2008. If you do not have the source folder on your PC after installing the update, try finding the folder named KB908002 in your system. If this does not help, just download the archive containing that folder at http://www.add-in-express.com/files/KB908002.zip.

The update shows an unpleasant dialog whenever you install your XLL; alas, you have to live with this.

Returning Dates from an XLL

Despite the restrictions introduced by internal context management in Excel (see Using the Excel Object Model in an XLL), some things are possible to do. Below is a sample (thank you, Thilo!) demonstrating the following aspects of XLL programming:

- Determining if Your UDF Is Called from the Insert Formula Dialog
- Determining What Cell / Worksheet / Workbook Your UDF Is Called From
- Returning Values When Your Excel UDF Is Called From an Array Formula
- Returning an Error Value from an Excel UDF
- It is safer to work with Excel in the `"en-US"` context. See also the following article on our technical blog - HowTo: Avoid “Old format or invalid type library” error.

To convert the code below to C#, call `ExcelApp.get_Range(callerAddress,Type.Missing)` instead of calling `ExcelApp.Range(callerAddress)` in VB.NET. Other changes are obvious.

```
...
Imports AddinExpress.MSO
Imports System.Threading
Imports System.Globalization
...
Public Shared Function GetCurrentDate() As Object
    If Not _Module.IsInFunctionWizard Then
        Dim caller As ADXExcelRef = _Module. _
            CallWorksheetFunction(ADXExcelWorksheetFunction.Caller)
        'returns [Book.xls]Sheet1!$A$1 or [Book.xls]Sheet1!$A$1:$B$2
        Dim callerAddress As String = caller.ConvertToA1Style
        Dim range As Excel.Range = _Module.ExcelApp.Range(callerAddress)
        Dim oldCultureInfo As CultureInfo = Thread.CurrentThread.CurrentCulture
        Thread.CurrentThread.CurrentCulture = New CultureInfo("en-US")
        range.NumberFormat = "mm/dd/yyyy"
        Thread.CurrentThread.CurrentCulture = oldCultureInfo
        If caller.ColumnFirst = caller.ColumnLast And _
            caller.RowFirst = caller.RowLast Then
            Return System.DateTime.Today.ToOADate()
        Else
            Dim v(2, 2) As Object
            v(0, 0) = "The current date is"
            v(0, 1) = System.DateTime.Today.ToOADate()
            v(1, 0) = "A sample error value)"
            v(1, 1) = ADXXlCVError.xlErrValue
            Return v
        End If
    Else
        Return "This UDF returns the current date."
    End If
End Function
```

Nevertheless, you should be very accurate when using this approach because the Excel Object Model doesn't expect such calls to be made when a formula is calculated. If you ever run into a problem with the code above, you can create a COM add-in that uses the SheetChange event in order to parse the formula just entered and format the corresponding cells as required.

COM Add-in, Excel UDF and AppDomain

It's very useful to combine an Excel add-in and a COM add-in (supporting Excel): the COM add-in can show controls that, for instance, provide some settings for your Excel UDF. To get the current state of the controls in your UDF, you use the ExcelApp.COMAddins property as shown in Accessing Public Members of Your COM Add-in from Another Add-in or Application. In the COM add-in, you can call any public method defined in your UDV via ExcelApp.Evaluate(...).

If you use both XLL module (ADXXLLModule) and add-in module (ADXAddinModule) in the same project, they are always loaded into the same AppDomain. But Excel Automation add-ins (ADXExcelAddinModule) are loaded into the default AppDomain **if you don't take any measures**. The need to have them in the same AppDomain can be caused by the necessity to share the same settings, for instance. To load the Automation add-in to the AppDomain of your COM add-in, you need to call any method of your Excel add-in using ExcelApp.Evaluate(...) **before** Excel (or the user) has a chance to invoke your Excel add-in. If such a call succeeds, your Excel Automation add-in is loaded into the AppDomain of your COM add-in.

The order in which Excel loads extensions is unpredictable; when the user installs another Excel add-in that order may change. We highly recommend testing your solutions with and without **Analysis Toolpak** installed. **Pay attention** that ExcelApp.Evaluate(...) returns a string value representing an error code if your UDF is still being loaded. In that case, you can try using several events to call your UDF: OnRibbonBeforeCreate, OnRibbonLoad, OnRibbonLoaded, AddinInitialize, AddinStartupComplete, as well as Excel-related events such as WindowActivate etc. We haven't tested, however, a scenario in which Excel refreshes a workbook containing formulas referencing an Excel Automation add-in. If you cannot win in such a scenario, you need to use an XLL add-in instead of the Automation one.

RTD

No RTD Servers in EXE

Add-in Express currently supports RTD Servers in DLLs only.

Update Speed for an RTD Server

Microsoft limits the minimal interval between updates to 2 seconds. There is a way to change this minimum value but Microsoft doesn't recommend doing this.

How to Get Actual Parameters of the RTD function When Using an Asterisk in the String## Properties of a Topic?

Strings passed to the RTD function allow identifying the topic. That is their only purpose. When there is no topic corresponding to the identifying strings, Add-in Express creates a new topic and passes it to the RefreshData event handler of the topic containing an asterisk (*). Therefore, you need to cast the sender argument to AddinExpress.RTD.ADXRTDTopic and get actual strings.

Inserting the RTD Function in a User-Friendly Way

The format of the RTD function (see Excel RTD Servers) isn't intuitive; the user prefers to call CurrentPrice("MSFT") rather than RTD("Stock.Quote", "", "MSFT", " Last").

You can do this by wrapping the RTD call in a UDF (thank you, Allan!). Note that calling the RTD function in a UDF makes Excel refresh the cell(s) automatically so you don't need to bother about this.

In your Excel Automation add-in, you use the RTD method provided by the Excel.WorksheetFunction interface:

```
Public Function CurrentPrice(ByVal topic1 As String) As Object
    Dim wsFunction As Excel.WorksheetFunction = ExcelApp.WorksheetFunction
    Dim result As Object = Nothing
    Try
        result = wsFunction.RTD("Stock.Quote", "", topic1, "Last")
    Catch
    Finally
        Marshal.ReleaseComObject(wsFunction)
    End Try
    Return result
End Function
```

To access an RTD server in your XLL add-in, you use the CallWorksheetFunction method provided by AddinExpress.MSO.ADXXLLModule. This method as well as the CallWorksheetCommand method is just a handy interface to functions exported by XLCALL32.DLL. Here is a sample

```
Public Shared Function CurrentPrice(ByVal topic1 As String) As Object
    If Not _Module.IsInFunctionWizard Then
        Return _Module. _
            CallWorksheetFunction( _
                ADXExcelWorksheetFunction.Rtd, _
                "Stock.Quote", _
                Nothing, _
                topic1, _
                "Last")
    Else
        Return "This UDF calls an RTD server."
    End If
End Function
```

Architecture

How to Develop the Modular Architecture of your COM and XLL Add-in?

Let's suppose that your COM add-in should conditionally provide (or not provide) some feature: let's call it MyFeature. You could create a class library project, add an ADXAddinAdditionalModule (see Add New Item dialog), and implement the feature.

Then you create a setup project that could, at your choice, either register the assembly using the vsdrpCOM option in the Register parameter of the assembly, or create appropriate keys in HKCU. Note that the former way may require the administrative privileges for the user. Now the class library can write the ProgID of the ADXAddinAdditionalModule into the app.config file of the add-in. When the add-in starts, it can read the app.config, create an ADXAddinAdditionalModuleItem and add it to the Modules collection of the ADXAddinModule class. The best place is the AddinInitialize event of the add-in module. For instance:

```
Friend WithEvents MyFeature As  _
   AddinExpress.MSO.ADXAddinAdditionalModuleItem

Private Sub AddinModule_AddinInitialize( _
   ByVal sender As Object, ByVal e As System.EventArgs) _
   Handles Me.AddinInitialize

   Dim MyFeatureProgId As String = System.Configuration. _
      ConfigurationManager.AppSettings("MyFeatureProgId")

   If MyFeatureProgId IsNot Nothing Then
      Me.MyFeature = _
         New AddinExpress.MSO.ADXAddinAdditionalModuleItem(Me.components)
      Me.MyFeature.ModuleProgID = MyFeatureProgId
      Me.Modules.Add(Me.MyFeature)
   End If
End Sub
```

If your ADXAddinAdditionalModule contains Ribbon controls, you will need to use the OnRibbonBeforeCreate event of the add-in module.

The same approach is applicable for XLL add-ins. Just use proper class types in the sample above.

Accessing Public Members of Your COM Add-in from Another Add-in or Application

You can access a public property or method defined in the add-in module via the following code path:

```
HostApp.COMAddins.Item({ProgID}).Object.MyPublicPropertyOrMethod(MyParameter)
```

The ProgID value above can be found in the ProgID attribute of the add-in module. Note that you access the MyPublicPropertyOrMethod above through late binding - see System.Type.InvokeMember. You can also find a number of samples in this document. And you can search our forums for more samples.

See also What is ProgID?

Finally

If your questions are not answered here, please see the HOWTOs section on our web site: see http://www.add-in-express.com/support/add-in-express-howto.php. You can also search our forums for an answer; the search page is http://www.add-in-express.com/forum/search.php. Another useful resource is our blog – see http://www.add-in-express.com/creating-addins-blog/.