Add-in Express™ for Microsoft® Office and .net

Developer's Guide

Revised on 2-Apr-15

Copyright © Add-in Express Ltd. All rights reserved.

Add-in Express, ADX Extensions, ADX Toolbar Controls, Afalina, AfalinaSoft and Afalina Software are trademarks or registered trademarks of Add-in Express Ltd. in the United States and/or other countries. Microsoft, Outlook, and the Office logo are trademarks or registered trademarks of Microsoft Corporation in the United States and/or other countries. Borland and the Delphi logo are trademarks or registered trademarks of Borland Corporation in the United States and/or other countries.

THIS SOFTWARE IS PROVIDED "AS IS" AND ADD-IN EXPRESS LTD. MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, ADD-IN EXPRESS LTD. MAKES NO REPRESENTATIONS OR WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE LICENSED SOFTWARE, DATABASE OR DOCUMENTATION WILL NOT INFRINGE ANY THIRD-PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.
# Table of Contents

Add-in Express™ for Microsoft® Office and .net ........................................... 2

Introduction .......................................................................................................... 9

Why Add-in Express? ............................................................................................ 10
  Add-in Express and Office Extensions .............................................................. 10
  Add-in Express Products ..................................................................................... 11
System Requirements .............................................................................................. 12
  Host Applications ................................................................................................ 12
Technical Support .................................................................................................... 13
Installing and Activating ........................................................................................ 14
  Activation Basics ................................................................................................. 14
  Setup Package Contents ...................................................................................... 15
  Solving Installation Problems ............................................................................ 15
Redistributables ....................................................................................................... 16

Getting Started ...................................................................................................... 17

Your First Microsoft Office COM Add-in .............................................................. 18
  A Bit of Theory .................................................................................................... 18
  Step #1 – Creating a COM Add-in Project .......................................................... 19
  Step #2 – Add-in Module .................................................................................... 23
  Step #3 – Add-in Module Designer ..................................................................... 24
  Step #4 – Adding a New Toolbar ......................................................................... 26
  Step #5 – Adding a New Toolbar Button ............................................................. 27
  Step #6 – Accessing Host Application Objects .................................................. 28
  Step #7 – Customizing Main Menus ..................................................................... 29
  Step #8 – Customizing Context Menus ............................................................... 31
  Step #9 – Handling Host Application Events ...................................................... 33
  Step #10 – Handling Excel Worksheet Events .................................................... 34
  Step #11 – Customizing the Ribbon User Interface .......................................... 36
  Step #12 – Adding Advanced Task Panes in Excel 2000-2013 .......................... 38
  Step #13 – Adding Advanced Task Panes in PowerPoint 2000-2013 ................ 41
  Step #14 – Adding Advanced Task Panes in Word 2000-2013 .......................... 44
  Step #15 – Running the COM Add-in ................................................................. 47
  Step #16 – Debugging the COM Add-in ............................................................. 49
  Step #17 – Deploying the COM Add-in .............................................................. 50
  What’s next? ....................................................................................................... 50

Your First Microsoft Outlook COM Add-in .......................................................... 51
  A Bit of Theory .................................................................................................... 51
  Step #1 – Creating a COM Add-in Project .......................................................... 52
  Step #2 – Add-in Module .................................................................................... 56
  Step #3 – Add-in Module Designer ..................................................................... 57
  Step #4 – Adding a New Explorer Toolbar .......................................................... 59
  Step #5 – Adding a New Toolbar Button ............................................................. 60
  Step #6 – Customizing the Outlook Ribbon UI .................................................. 61
  Step #7 – Adding a New Inspector Toolbar ......................................................... 62
  Step #8 – Customizing Main Menu in Outlook 2000-2007 .................................. 64
Step #9 – Customizing Outlook Context Menus ............................................................... 66
Step #10 – Adding an Advanced Outlook Region in Outlook 2000-2013 ........................................... 69
Step #11 – Accessing Outlook Objects ........................................................................... 72
Step #12 – Handling Outlook Events ............................................................................. 74
Step #13 – Handling Events of Outlook Items Object .................................................... 75
Step #14 – Adding Property Pages to the Folder Properties Dialog ........................ 77
Step #15 – Intercepting Keyboard Shortcuts .................................................................. 80
Step #16 – Running the COM Add-in .............................................................................. 80
Step #17 – Debugging the COM Add-in ......................................................................... 83
Step #18 – Deploying the COM Add-in .......................................................................... 84
What’s next? ....................................................................................................................... 84

Your First Excel RTD Server .......................................................................................... 85
A Bit of Theory .................................................................................................................... 85
Step #1 – Creating an RTD Server Project .................................................................... 86
Step #2 – RTD Server Module ....................................................................................... 89
Step #3 – RTD Server Module Designer ....................................................................... 89
Step #4 – Adding and Handling a New Topic ................................................................. 90
Step #5 – Running the RTD Server ................................................................................ 92
Step #6 – Debugging the RTD Server .............................................................................. 93
Step #7 – Deploying the RTD Server ............................................................................ 94
What’s next? ....................................................................................................................... 94

Your First Smart Tag ....................................................................................................... 95
A Bit of Theory .................................................................................................................... 95
Step #1 – Creating a Smart Tag Library Project ............................................................ 96
Step #2 – Smart Tag Module ......................................................................................... 99
Step #3 – Smart Tag Module Designer ....................................................................... 99
Step #4 – Adding a New Smart Tag ............................................................................. 101
Step #5 – Adding Smart Tag Actions ......................................................................... 102
Step #6 – Running the Smart Tag ................................................................................. 103
Step #7 – Debugging the Smart Tag ............................................................................. 104
Step #8 – Deploying the Smart Tag ............................................................................. 105
What’s next? ....................................................................................................................... 105

Your First Excel Automation Add-in ........................................................................... 106
A Bit of Theory .................................................................................................................... 106
Step #1 – Creating a COM Add-in Project .................................................................. 107
Step #2 – Adding a COM Excel Add-in Module ........................................................... 112
Step #3 – Writing a User-Defined Function ................................................................. 113
Step #4 – Running the Add-in ..................................................................................... 113
Step #5 – Debugging the Excel Automation Add-in ..................................................... 114
Step #6 – Deploying the Add-in .................................................................................. 115
What’s next? ....................................................................................................................... 115

Your First XLL Add-in .................................................................................................... 116
A Bit of Theory .................................................................................................................... 116
Step #1 – Creating an XLL Add-in Project .................................................................. 117
Step #2 – XLL Module .................................................................................................... 120
Step #3 – Creating a User-Defined Function ............................................................... 120
Step #4 – Configuring UDFs ........................................................................................ 121
Step #5 – Running the XLL Add-in .............................................................................. 126
Step #6 – Debugging the XLL Add-in .......................................................................... 127
Step #7 – Deploying the XLL Add-in ........................................................................... 128
What’s next? ....................................................................................................................... 128
# Add-in Express Components

## Add-in Express Modules

Table of Contents

<table>
<thead>
<tr>
<th>Add-in Express Modules</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Office Ribbon UI Components</td>
<td>130</td>
</tr>
<tr>
<td>How Ribbon Controls Are Created?</td>
<td>131</td>
</tr>
<tr>
<td>Referring to Built-in Ribbon Controls</td>
<td>132</td>
</tr>
<tr>
<td>Intercepting Built-in Ribbon Controls</td>
<td>133</td>
</tr>
<tr>
<td>Disabling Built-in Ribbon Controls</td>
<td>134</td>
</tr>
<tr>
<td>Positioning Ribbon Controls</td>
<td>134</td>
</tr>
<tr>
<td>Images on Ribbon Controls</td>
<td>134</td>
</tr>
<tr>
<td>Creating Ribbon Controls at Run Time</td>
<td>135</td>
</tr>
<tr>
<td>Updating Ribbon Controls at Run Time</td>
<td>135</td>
</tr>
<tr>
<td>Determining a Ribbon Control’s Context</td>
<td>136</td>
</tr>
<tr>
<td>Sharing Ribbon Controls across Multiple Add-ins</td>
<td>136</td>
</tr>
<tr>
<td>CommandBar UI Components</td>
<td>145</td>
</tr>
<tr>
<td>Toolbar</td>
<td>146</td>
</tr>
<tr>
<td>Main Menu</td>
<td>147</td>
</tr>
<tr>
<td>Context Menu</td>
<td>148</td>
</tr>
<tr>
<td>Outlook Toolbars and Main Menus</td>
<td>149</td>
</tr>
<tr>
<td>Connecting to Existing Command Bars</td>
<td>150</td>
</tr>
<tr>
<td>Connecting to Existing CommandBar Controls</td>
<td>150</td>
</tr>
<tr>
<td>How Command Bars and Their Controls Are Created and Removed</td>
<td>151</td>
</tr>
<tr>
<td>Command Bars in the Ribbon UI</td>
<td>152</td>
</tr>
<tr>
<td>Command Bar Control Properties and Events</td>
<td>152</td>
</tr>
<tr>
<td>Command Bar Control Types</td>
<td>153</td>
</tr>
<tr>
<td>Custom Task Panes in Office 2007-2013</td>
<td>154</td>
</tr>
<tr>
<td>Advanced Outlook Regions and Advanced Office Task Panes</td>
<td>157</td>
</tr>
<tr>
<td>Introducing Advanced Task Panes in Word, Excel and PowerPoint</td>
<td>157</td>
</tr>
<tr>
<td>Introducing Advanced Outlook Form and View Regions</td>
<td>158</td>
</tr>
<tr>
<td>The UI Mechanics</td>
<td>164</td>
</tr>
<tr>
<td>Advanced Excel Task Panes</td>
<td>171</td>
</tr>
<tr>
<td>Advanced Outlook Regions</td>
<td>173</td>
</tr>
<tr>
<td>Events</td>
<td>180</td>
</tr>
<tr>
<td>Application-level Events</td>
<td>180</td>
</tr>
<tr>
<td>Events Classes</td>
<td>180</td>
</tr>
<tr>
<td>Intercepting Keyboard Shortcuts</td>
<td>181</td>
</tr>
<tr>
<td>Outlook UI Components</td>
<td>182</td>
</tr>
<tr>
<td>Outlook Bar Shortcut Manager</td>
<td>182</td>
</tr>
<tr>
<td>Outlook Property Page</td>
<td>182</td>
</tr>
<tr>
<td>Other Components</td>
<td>183</td>
</tr>
<tr>
<td>Smart Tag</td>
<td>183</td>
</tr>
<tr>
<td>RTD Topic</td>
<td>183</td>
</tr>
<tr>
<td>Custom Toolbar Controls</td>
<td>184</td>
</tr>
<tr>
<td>What is ADXCommandBarAdvancedControl</td>
<td>184</td>
</tr>
<tr>
<td>Hosting any .NET Controls</td>
<td>184</td>
</tr>
<tr>
<td>Control Adapters</td>
<td>185</td>
</tr>
<tr>
<td>ADXCommandBarAdvancedControl</td>
<td>186</td>
</tr>
<tr>
<td>Application-specific Control Adapters</td>
<td>187</td>
</tr>
<tr>
<td>Your First .NET Control on an Office Toolbar</td>
<td>188</td>
</tr>
</tbody>
</table>
Deploying Office Extensions

All Deployment Technologies at a Glance

Deployment: Things to Consider

- How Your Office Extension Is Registered
- How Your Office Extension Loads Into an Office Application
- Per-user or Per-machine?
- Installing and Registering
- Permissions Required
- Files to Deploy
- Publishing from the Command Prompt

Creating MSI Installers

- Running the Setup Project Wizard
- Creating a Visual Studio Setup Project Manually
- Creating a WIX Project

ClickOnce Deployment

- ClickOnce Overview
- Add-in Express ClickOnce Solution
- Customizing ClickOnce installations

ClickTwice :) Deployment

- Introduction to ClickTwice
- Publishing with ClickTwice :)
- Files Generated by ClickTwice :)
- Updating an Office Extension via ClickTwice :)
- Customizing ClickTwice installations
- Step-by-step Samples

Deployment Step-by-steps

- Deploying a per-user Office extension via an MSI installer
- Deploying a per-machine Office extension via an MSI installer
- Deploying a per-user Office extension via Group Policy
- Deploying a per-user Office extension via ClickOnce
- Updating a per-user Office extension via ClickOnce
- Deploying an Office extension via ClickTwice :)
- Updating an Office extension via ClickTwice :)

Tips and Notes

Recommended Blogs and Videos

- On COM Add-ins, Ribbon and CommandBar UI
- On Excel User-Defined Functions (UDFs)
- On Excel RTD Server
- On Deployment
- On Outlook Development
- On Excel Development
- On Advanced Regions and Advanced Task Panes

Office UI Options for Developers

- Installed Add-ins
- COM Add-ins dialog
- Disabled Items dialog
- Excel Add-ins dialog
- Get Informed about Errors in Ribbon markup
## Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable all Application Add-ins</td>
<td>306</td>
</tr>
<tr>
<td>Require application add-ins to be signed</td>
<td>307</td>
</tr>
<tr>
<td><strong>Development Tips</strong></td>
<td></td>
</tr>
<tr>
<td>Getting Help on COM Objects, Properties and Methods</td>
<td>309</td>
</tr>
<tr>
<td>Developing an Add-in Supporting Several Office Versions</td>
<td>309</td>
</tr>
<tr>
<td>Choosing Interop Assemblies</td>
<td>309</td>
</tr>
<tr>
<td>Use the Latest version of the Loader</td>
<td>310</td>
</tr>
<tr>
<td>Several Office Versions on the Machine</td>
<td>310</td>
</tr>
<tr>
<td>How to Find Files on the Target Machine Programmatically?</td>
<td>311</td>
</tr>
<tr>
<td>Configuring an Add-in</td>
<td>311</td>
</tr>
<tr>
<td>Using Threads</td>
<td>311</td>
</tr>
<tr>
<td>Releasing COM Objects</td>
<td>311</td>
</tr>
<tr>
<td>Wait a Little</td>
<td>312</td>
</tr>
<tr>
<td>What is ProgID?</td>
<td>314</td>
</tr>
<tr>
<td><strong>COM Add-in Tips</strong></td>
<td>315</td>
</tr>
<tr>
<td>Delays at Add-in Start-up</td>
<td>315</td>
</tr>
<tr>
<td>FolderPath Property Is Missing in Outlook 2000 and XP</td>
<td>315</td>
</tr>
<tr>
<td>Word Add-ins, Command bars, and normal.dot</td>
<td>316</td>
</tr>
<tr>
<td>If you use an Express edition of Visual Studio</td>
<td>317</td>
</tr>
<tr>
<td><strong>Command Bars and Controls Tips</strong></td>
<td>318</td>
</tr>
<tr>
<td>CommandBar Terminology</td>
<td>318</td>
</tr>
<tr>
<td>ControlTag vs. Tag Property</td>
<td>318</td>
</tr>
<tr>
<td>Pop-ups</td>
<td>318</td>
</tr>
<tr>
<td>Built-in Controls and Command Bars</td>
<td>318</td>
</tr>
<tr>
<td>CommandBar.Position = adxMsoBarPopup</td>
<td>319</td>
</tr>
<tr>
<td>Built-in and Custom Command Bars in Ribbon-enabled Office Applications</td>
<td>319</td>
</tr>
<tr>
<td>Transparent Icon on a CommandBarButton</td>
<td>319</td>
</tr>
<tr>
<td>Navigating Up and Down the Command Bar System</td>
<td>320</td>
</tr>
<tr>
<td>Hiding and Showing Outlook Command Bars</td>
<td>320</td>
</tr>
<tr>
<td><strong>Debugging and Deploying Tips</strong></td>
<td>321</td>
</tr>
<tr>
<td>Breakpoints are Not Hit When Debugging</td>
<td>321</td>
</tr>
<tr>
<td>Don't Use Message Boxes When Debugging</td>
<td>321</td>
</tr>
<tr>
<td>Conflicts with Office Extensions Developed in .NET Framework 1.1</td>
<td>321</td>
</tr>
<tr>
<td>How to find if Office 64-bit is installed on the target machine</td>
<td>322</td>
</tr>
<tr>
<td>Updating on the Fly</td>
<td>322</td>
</tr>
<tr>
<td>For All Users or For the Current User?</td>
<td>323</td>
</tr>
<tr>
<td>User Account Control (UAC) on Vista, Windows 7 and Windows Server 2008</td>
<td>324</td>
</tr>
<tr>
<td>Deploying Word Add-ins</td>
<td>324</td>
</tr>
<tr>
<td>InstallAllUsers Property of the Setup Project</td>
<td>324</td>
</tr>
<tr>
<td>Deploying – Shadow Copy</td>
<td>324</td>
</tr>
<tr>
<td>Deploying – “Everyone” Option in a COM Add-in MSI package</td>
<td>325</td>
</tr>
<tr>
<td>Deploying Office Extensions</td>
<td>325</td>
</tr>
<tr>
<td>ClickOnce Application Cache</td>
<td>325</td>
</tr>
<tr>
<td>ClickOnce Deployment</td>
<td>325</td>
</tr>
<tr>
<td>Custom Actions When Your COM Add-in Is Uninstalled</td>
<td>326</td>
</tr>
<tr>
<td>Bypassing the AlwaysInstallElevated Policy</td>
<td>326</td>
</tr>
<tr>
<td><strong>Excel UDF Tips</strong></td>
<td>327</td>
</tr>
<tr>
<td>What Excel UDF Type to Choose?</td>
<td>327</td>
</tr>
<tr>
<td>My Excel UDF Doesn't Work</td>
<td>328</td>
</tr>
<tr>
<td>My XLL Add-in Doesn't Show Descriptions</td>
<td>328</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Can an Excel UDF Return an Object of the Excel Object Model?</td>
<td>329</td>
</tr>
<tr>
<td>Why Using a Timer in an XLL isn't Recommended?</td>
<td>329</td>
</tr>
<tr>
<td>Parameterless UDFs</td>
<td>330</td>
</tr>
<tr>
<td>Can an Excel UDF Modify Multiple Cells?</td>
<td>330</td>
</tr>
<tr>
<td>Can an Excel UDF Return an Empty Cell?</td>
<td>331</td>
</tr>
<tr>
<td>Using the Excel Object Model in an XLL</td>
<td>331</td>
</tr>
<tr>
<td>Determining What Cell / Worksheet / Workbook Your UDF Is Called From</td>
<td>331</td>
</tr>
<tr>
<td>Determining if Your UDF Is Called from the Insert Formula Dialog</td>
<td>331</td>
</tr>
<tr>
<td>Returning an Error Value from an Excel UDF</td>
<td>332</td>
</tr>
<tr>
<td>XLL and Shared Add-in Support Update</td>
<td>332</td>
</tr>
<tr>
<td>Returning Values When Your Excel UDF Is Called From an Array Formula</td>
<td>332</td>
</tr>
<tr>
<td>Returning Dates from an XLL</td>
<td>333</td>
</tr>
<tr>
<td>Multi-threaded XLLs</td>
<td>334</td>
</tr>
<tr>
<td>Asynchronous XLLs</td>
<td>335</td>
</tr>
<tr>
<td>COM Add-in, Excel UDF and AppDomain</td>
<td>336</td>
</tr>
<tr>
<td>RTD Tips</td>
<td>337</td>
</tr>
<tr>
<td>No RTD Servers in EXE</td>
<td>337</td>
</tr>
<tr>
<td>Update Speed for an RTD Server</td>
<td>337</td>
</tr>
<tr>
<td>Inserting the RTD Function in a User-Friendly Way</td>
<td>337</td>
</tr>
<tr>
<td>Troubleshooting</td>
<td>338</td>
</tr>
<tr>
<td>Troubleshooting add-in registration</td>
<td>338</td>
</tr>
<tr>
<td>Troubleshooting add-in loading</td>
<td>339</td>
</tr>
<tr>
<td>Architecture Tips</td>
<td>344</td>
</tr>
<tr>
<td>Developing Multiple Office Extensions in the Same Project</td>
<td>344</td>
</tr>
<tr>
<td>How to Develop the Modular Architecture of your COM and XLL Add-in</td>
<td>344</td>
</tr>
<tr>
<td>Accessing Public Members of Your COM Add-in from Another Add-in or Application</td>
<td>345</td>
</tr>
<tr>
<td>Finally</td>
<td>346</td>
</tr>
</tbody>
</table>
Add-in Express is a development tool designed to simplify and speed up the development of Office COM Add-ins, Real-Time Data servers (RTD servers), Smart Tags, Excel Automation Add-ins and Excel XLL add-ins in Visual Studio 2008-2013 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, insulated, managed, deployable, and updatable Office extensions.

- Managed Office Extensions

You develop them in every programming language available for Visual Studio (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 2013. See [Choosing Interop Assemblies](#).

- Deployable and updatable Office Extensions

Add-in Express generates 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 Office Extensions](#).
Add-in Express Products

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

- Add-in Express for Microsoft Office and CodeGear VCL


- Add-in Express for Internet Explorer and .NET


- Security Manager for Microsoft Outlook

This is a product designed for Outlook solution developers. It allows controlling the Outlook email 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


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 declared deprecated in Office 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](http://www.add-in-express.com) provides a wealth of information and software downloads for Add-in Express developers, including:

- Our [technical blog](#) provides the 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](#) helps finding 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 use our [forums](#) or email us at support@add-in-express.com. Urgent problems can be reported by email or using the [Escalate To](#) buttons on the forums.
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 activation process, the activation 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 license 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 license key and a hardware identifier are the only pieces of information required to activate the product. No other information is collected from your PC or sent to the activation server.

If you choose the Automatic Activation 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 code using this information and sends it back to the activation wizard. The wizard saves the activation code to the registry.

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

Activation is completely anonymous; no personally identifiable information is required. The activation code 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 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 contains 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 - 8 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. If this is applicable, run setup.exe by right-clicking it and choosing “Run as administrator” in the context menu.

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:

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddinExpress.MSO.2005.dll</td>
<td>Office + XLL add-ins + Excel Automation add-ins</td>
</tr>
<tr>
<td>AddinExpress.RTD.2005.dll</td>
<td>RTD servers</td>
</tr>
<tr>
<td>AddinExpress.SmartTag.2005.dll</td>
<td>Smart tags</td>
</tr>
<tr>
<td>AddinExpress.OL.2005.dll</td>
<td>Advanced Outlook form regions</td>
</tr>
<tr>
<td>AddinExpress.PP.2005.dll</td>
<td>Advanced Office task panes in PowerPoint</td>
</tr>
<tr>
<td>AddinExpress.WD.2005.dll</td>
<td>Advanced Office task panes in Word</td>
</tr>
<tr>
<td>AddinExpress.XL.2005.dll</td>
<td>Advanced Office task panes in Excel</td>
</tr>
<tr>
<td>AddinExpress.ToolbarControls.2005.dll</td>
<td>.NET controls on Office command bars</td>
</tr>
<tr>
<td>AddinExpress.Deployment.dll</td>
<td>Support for ClickTwice – MSI-based web deployment technology, see <a href="#">ClickTwice :) Deployment</a></td>
</tr>
</tbody>
</table>
Getting Started

Here 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 logic
- 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

These are the sample projects described in this chapter:

- Your First Microsoft Office COM Add-in
- Your First Microsoft Outlook COM Add-in
- Your First Excel RTD Server
- Your First Smart Tag
- Your First Excel Automation Add-in
- Your First XLL Add-in

You can download the projects (C# and VB.NET) here; the download link is labeled Add-in Express for Office and .NET sample projects.
Your First Microsoft Office COM Add-in

The sample project below demonstrates how you create a COM add-in supporting several Office applications: Excel, Word and PowerPoint. The add-in creates a custom toolbar and adds a CommandBar button to the toolbar, main menu (in Office 2000-2003) and context menu, and a Ribbon button to the Ribbon UI of Office 2007-2013. Also, the add-in creates an advanced task pane supporting versions 2000-2013 of the host applications. The source code of the project – both VB.NET and C# versions – can be downloaded [here](#); the download link is labeled Add-in Express for Office and .NET sample projects.

A Bit of Theory

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).

COM add-ins is the only way to add 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 advanced task panes in Office 2000-2013. In a COM add-in targeting Outlook, you can add custom option pages to the Tools | Options and Folder Properties dialogs. 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 email when it is being sent; it can cancel saving an Excel workbook or it can check if a Word document or selected text meets some requirements.

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. Add-in Express generates a per-user add-in project; your add-in is per-machine if the add-in module has ADXAddinModule.RegisterForAllUsers = True. Registering for all users means creating registry entries in HKLM and that means the user registering a per-machine add-in must have administrative permissions. Accordingly, RegisterForAllUsers = False means writing to HKCU (=for the current user). See [Registry Keys](#).

*Before you modify the RegisterForAllUsers property, you must unregister the add-in project on your development PC and make sure that adxloder.dll.manifest is writable.*

A standard user may turn a per-user add-in off and on in the COM Add-ins dialog. You use that dialog to check if your add-in is active.
Step #1 - Creating a COM Add-in Project

Make sure that you have administrative permissions before running Visual Studio. Also, if you have Windows Vista, Windows 7, Windows Server 2008 or Windows 8, run Visual Studio via Run as Administrator.

In Visual Studio, open the New Project dialog and navigate to the Extensibility folder.

Choose Add-in Express COM Add-in and click OK.
This starts the COM Add-in project wizard. It allows choosing a programming language and specifying the oldest Office version your add-in needs to support.

Choosing a particular Office version will add corresponding interop assemblies to the project. Later on, you can replace interop assemblies in your project. If you are in doubt, choose Microsoft Office 2000 as the minimum supported Office version. If you need background information, see Choosing Interop Assemblies.

Choose your programming language and the minimum Office version you need to support and click Next.
The wizard allows creating add-in projects targeting several Office applications.

You can choose one or more Microsoft Office applications in the above window. For every chosen application, the project wizard will do the following:

- copy the corresponding interop assembly to the Interops folder of your project folder,
- add an assembly reference to the project
- add a COM add-in module to the project
- set up the SupportedApp property of the add-in module
- add a property to the add-in module; the property name reflects the name of the chosen Office application; you will use that property to access the object model of the Office application loading your add-in, see Step #6 – Accessing Host Application Objects.

See also Developing an Add-in Supporting Several Office Versions.

Select which Office applications will be supported by your add-in and click Next.
The wizard allows specifying a strong name file; it is used to uniquely identify your add-in assembly.

If you don't have a specific strong name key file, choose Generate new. If you are in doubt, choose Generate new. If, later on, you need to use a strong name key file, specify its name on the Signing tab of your project properties. You are required to unregister your add-in project before using another strong name.

Choose Generate new or specify an existing .snk file and click Finish.
The project wizard creates and opens a new solution in the IDE.

![Image of Visual Studio IDE with AddinModule.vb file]

The solution contains an only project, the COM add-in project. The add-in project contains the AddinModule.vb (or AddinModule.cs) file discussed in the next step.

**Step #2 - Add-in Module**

AddinModule.vb (or AddinModule.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 configure the add-in's settings 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 the Solution Explorer, right-click AddinModule.vb (or
Add-in Express for Office and .net

Your First Microsoft Office COM Add-in

AddinModule.cs) and choose View Code in the popup menu.

In the code of the module, pay attention to three points:

1. The comment line in the constructor of the module

The text suggests that you write initialization code in the AddinInitialize or AddinStratupComplete events of the add-in module, not in the constructor. This is important because an instance of the module is created (this runs the constructor) when your add-in is being registered and unregistered. When the module is created for this purpose, no Office objects, properties, methods or events are available. If your code is not prepared for this, the installer of your add-in will fail.

Moving custom initialization code out of the module's constructor allows preventing installation failures.

2. The ExcelApp, WordApp, and PowerPointApp properties

You use these properties – they are generated by the COM add-in project wizard – as entry points to the object models of the corresponding Office applications; see Step #6 – Accessing Host Application Objects.

3. The CurrentInstance property

This property returns the current instance of the add-in module, a very useful thing if you need to access the add-in module from e.g. a task pane.

Let's look at the module designer.

Step #3 - Add-in Module Designer

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

The add-in module designer view provides access to the following four areas shown in the screenshot below:

1. Add-in Express Toolbox – (#1 in the screenshot below) it contains commands; clicking a command creates a new Add-in Express component and places it onto the add-in module;

2. Add-in module designer – (#2 in the screenshot below) it is a usual designer;
3. **In-place designer** – (#3 in the screenshot below) if there’s a visual designer for the currently selected Add-in Express component, then it is shown in this area;
4. **Help panel** – see #4 in the screenshot below.

To add any Add-in Express component onto the module, you click an appropriate command in the Add-in Express Toolbox (#1 in the screenshot above). Then you select the newly added component and set it up using the visual designer (#3 in the screenshot above) and Properties window. This procedure is demonstrated in the sections below.

Note that the add-in module provides a number of properties. Click the designer surface and specify the name and description of your add-in in the Properties window. Also, pay attention to the `RegisterForAllUsers` property – it is essential for distinguishing per-user and per-machine COM add-ins; you must unregister the add-in project before modifying this property.
Step #4 - Adding a New Toolbar

In the Add-in Express Toolbox, click the Add ADXCommandBar button. This places a new ADXCommandBar component onto the add-in module. Select the just added ADXCommandBar and specify the toolbar name in the CommandBarName property (see in the Properties window).

You may use both CommandBar UI Components and Office Ribbon UI Components on 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.
ADXCommandBar creates controls on the toolbar the name of which you specify in the CommandBarName property. If the specified toolbar is missing on the host application, ADXCommandBar creates it. That is, if you add some controls to the ADXCommandBar which CommandBarName is set to "Standard", this will create the controls on the built-in Standard toolbar, while specifying CommandBarName = "Standard2" will create a new toolbar, Standard2, with the controls on it. If the Standard2 toolbar exists in the host application, the buttons will be added to that toolbar. You can download Built-in Controls Scanner and use it to get the names of command bars in Office 2000-2013 applications.

Step #5 - Adding a New Toolbar Button

Select the command bar component on the add-in module and open the in-place designer area. In this area, you see the visual designer of the ADXCommandBar component. Use the visual designer to add a toolbar button onto the command bar component.
To achieve the result shown in the screenshot above, select the new button and open the Properties window where you specify the button's Caption property and add an event handler to the Click event. Also, in the screenshot above, we demonstrate the properties that make the icon visible and transparent: Image, ImageList, and ImageTransparentColor. Please pay attention: to get the icon displayed, the Style property (not visible in the screenshot) is set to adxMsoButtonIconAndCaption. This is because command bar buttons do not show icons by default. To get the icon transparent, the value of the ImageTransparentColor property is chosen in accordance with the bitmap used. See also Command Bar Control Properties and Events.

**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. The project wizard adds properties such as ExcelApp, WordApp, etc. to the module; the properties just cast HostApplication to the appropriate type. You use these properties as entry points to the corresponding object models:

```vba
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 ActiveWindow IsNot 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 As String = Words.Count.ToString()
                Marshal.ReleaseComObject(Selection)
                Marshal.ReleaseComObject(Range)
                Marshal.ReleaseComObject(Words)
                Result = "There are " + WordCountString + " words currently selected"
        End Select
    End If
```

Add-in Express for Office and .net

Your First Microsoft Office COM Add-in
**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, _Me.HostApplication.ActiveWindow()_ is wrapped in a try/catch block: this prevents a notorious exception which occurs when you call _ActiveWindow()_ on the _Word.Application_ object and there are no documents opened in Word. Second, you have to release every COM object created in your code, see [Releasing COM Objects](#) for more details.

**Step #7 - Customizing Main Menus**

Add-in Express provides a component to customize the main menu of a pre-Ribbon 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, starting from Excel 2007, these menus are replaced with the Ribbon UI. Nevertheless, the CommandBar objects representing the main menus are still accessible programmatically and you may want to use this fact in your code. As for customizing main menus in Outlook, see [Step #8 – Customizing Main Menu in Outlook 2000-2007](#).

In this sample, we are going to customize the _File_ menu in Excel and Word, version 2000-2003. You start with adding two _ADXMainMenu_ components (from the Add-in Express Toolbox) onto the add-in module and specifying correct host applications in their _SupportedApp_ properties. Then, in the _CommandBarName_ property, you specify the main menu you want to customize.
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.

In the screenshot above, you see the ADXMainMenu component and its visual designer. You use the visual designer to populate the ADXMainMenu with controls. Note that only command bar buttons and command bar popups can be added onto a main menu.

To add a custom control to the File menu, you add a popup control to the ADXMainMenu component 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 a built-in popup with the corresponding ID. And 30002 is the ID of the File menu item in pre-Ribbon Office applications.

To find this and similar IDs, use our free Built-in Control Scanner. For background info please see Connecting to Existing CommandBar Controls.
Now you add a button and set its properties in the way described in Step #5 – Adding a New Toolbar Button.

Pay attention to the **BeforeID** property of the button component. To place the button before the built-in *Save* button, you set this property to 3, which is the ID of the *Save* button. 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*.

See also Step #11 – Customizing the Ribbon User Interface for customizing the Office button menu in Office 2007 and the File tab (Backstage View) in Office 2010-2013. Find more details in CommandBar UI Components and Office Ribbon UI Components.

**Step #8 - Customizing Context Menus**

Add-in Express allows customizing CommandBar-based context menus in Office 2000-2013 with the ADXContextMenu component, find the corresponding command on the Add-in Express Toolbox. Its use is similar to that of the ADXMainMenu component. See how to set up such a component in order 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, a context menu of which you need to customize
- Specify the name of the context menu to customize
- Use the in-place designer to add custom controls to the *Controls* collection of the component

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: use 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 Ribbon-based context menus of Office 2010-2013: a custom context menu item created with the ADXContextMenu component will always be shown below any built-in and custom context menu items in a Ribbon-based context menu.
To add a custom item to a Ribbon-based context menu (Office 2010-2013), 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 `ContextMenuNames` property of the component.

See also Step #11 – Customizing the Ribbon User Interface, CommandBar UI Components and Office Ribbon UI Components.
Step #9 - Handling Host Application Events

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

With the event components, you handle 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:

```vba
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 Class* item in the *Add New Item* dialog in Visual Studio. A descendant of this class allows connecting to an *Excel.Worksheet* object and handling its events. You can use it for creating a set(s) of business rules for handling specific Excel worksheets.

Selecting the *Excel Worksheet Events Class* item and clicking *OK* adds an event class to your project (see the screenshot to the right).

In the event class, add the following code to the procedure handling the *BeforeRightClick* event of the *Worksheet* class:

```vbnet
Selecting the Excel Worksheet Events Class item and clicking OK adds an event class to your project (see the screenshot to the right).

In the event class, add the following code to the procedure handling the BeforeRightClick event of the Worksheet class:
```
Public Overrides Sub ProcessBeforeRightClick(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(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(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, choose the *Add Ribbon Tab* command on the Add-in Express Toolbox; it places a new `ADXRibbonTab` component onto the module. Select the component and, in the in-place designer, use toolbar buttons or context menu to add or delete components that create 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*. Finally, you select the group, add a button and set its caption to *My Ribbon Button*. Use the `ImageList`, `ImageTransparentColor`, `Image` and `Glyph` properties to set the image for the button. See also [Images on Ribbon Controls](#).

Now add an event handler to the *Click* event of the button and write the following code:

```vbnet
Private Sub AdxRibbonButton1_OnClick(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles AdxRibbonButton1.Click

    ' Your code here

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

The sample project described here creates a custom Ribbon tab. You can also customize a built-in Ribbon tab, please check Referring to Built-in Ribbon Controls.

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 AdxRibbonOfficeMenu. As to the Backstage View, also known as the File Tab, the sample project provides the AdxBackstageView component that implements the customization shown in Figure 3 at Introduction to the Office 2010 Backstage View for Developers. Note that if you customize the Office Button menu only Add-in Express maps your controls to the Backstage View if Office 2010 or 2013 loads the add-in. 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 Office Ribbon UI Components.
Step #12 - Adding Advanced Task Panes in Excel 2000-2013

Creating a new Excel task pane includes the following steps:

- Use the Add-in Express Toolbox to add an Excel Task Panes Manager, `ADXExcelTaskPanesManager`, onto the add-in module.
- Open the Add New Item dialog in Visual Studio to add an Add-in Express Excel Task Pane, `ADXExcelTaskPane`, to the project.

- Select the Excel Task Panes Manager component and add an item to its **Items** collection (see the screenshot below).

![Add ADXExcelTaskPanesManager](imageURL)

![Add New Item - MyAddin1](imageURL)
Select the newly added item and, in the Properties window, specify properties as follows:

- **AlwaysShowHeader** - specifies that the pane header will be shown even if the pane is the only pane in the current region (optional).
- **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 (optional).
- **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. In this add-in, we use Position=Bottom.
- **TaskPaneClassName** - specifies the class name of the Excel task pane.
Now you add a label onto the form and set its caption in the following code:

```vbscript
Private Sub RefreshTaskPane()
    Select Case Me.HostName
    Case "Excel"
        Dim Pane As ADXExcelTaskPanel1 = _
            TryCast(Me.AdxExcelTaskPanesCollectionItem1.TaskPaneInstance, _
                ADXExcelTaskPanel1)
        If Pane IsNot Nothing Then Pane.Label1.Text = Me.GetInfoString()
    Case "Word", "PowerPoint"
    Case Else 'MessageBox.Show("Invalid host application!")
    End Select
End Sub
```

See also Advanced Outlook Regions and Advanced Office Task Panes and Advanced Excel Task Panes.
Step #13 - Adding Advanced Task Panes in PowerPoint 2000-2013

Now you add a PowerPoint task pane:

- Use the Add-in Express Toolbox to add a PowerPoint Task Panes Manager, ADXPowerPointTaskPanesManager, onto the add-in module.
- Open the Add New Item dialog in Visual Studio to add an Add-in Express PowerPoint Task Pane, ADXPowerPointTaskPane, to the project.

- Select the PowerPoint Task Panes Manager component and add an item to its Items collection.
Select the newly added item and, in the *Properties* window, specify properties as follows:

- **AlwaysShowHeader** - specifies that the pane header will be shown even if the pane is the only pane in the current region (optional).
- **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 (optional).
- **Position** - specifies the region in which an instance of the pane will be shown. PowerPoint panes are allowed in four regions docked to the four sides of the main PowerPoint window: *Right*, *Bottom*, *Left*, and *Top*. The fifth region is *Unknown*. In this add-in, we use *Position=Bottom*.
- **TaskPaneClassName** - specifies the class name of the PowerPoint task pane.
Now add a label onto the form, write a property that reads and updates the label, and modify `RefreshTaskPane` in order to set the property value:

```vbnet
Private Sub RefreshTaskPane()
    Select Case Me.HostName
        Case "Excel", "Word"
            Case "PowerPoint"
                Dim Pane As ADXPowerPointTaskPane1 = _
                    TryCast(Me.AdxPowerPointTaskPanesCollectionItem1.TaskPaneInstance, _
                        ADXPowerPointTaskPane1)
                If Pane IsNot Nothing Then Pane.Label1.Text = Me.GetInfoString()
    End Select
End Sub
```

See also [Advanced Outlook Regions and Advanced Office Task Panes](https://www.add-in-express.com).
Step #14 - Adding Advanced Task Panes in Word 2000-2013

You add a Word task pane in the same manner:

- Use the Add-in Express Toolbox to add an `ADXWordTaskPanesManager` to your add-in module.
- Open the Add New Item dialog in Visual Studio to add an Add-in Express Word Task Pane (`ADXWordTaskPane`) to the project.

- Select the Word Task Panes Manager component and add an item to its `Items` collection.
Select the newly added item and, in the Properties window, specify properties as follows:

- **AlwaysShowHeader** - specifies that the pane header will be shown even if the pane is the only pane in the current region (optional).
- **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 (optional).
- **Position** - specifies the region in which an instance of the pane will be shown. Word panes are allowed in four regions docked to the four sides of the main Word window: Right, Bottom, Left, and Top. The fifth region is Unknown. In this add-in, we use Position=Bottom.
- **TaskPaneClassName** - specifies the class name of the Word task pane.
Now add a label onto the form and update `RefreshTaskPane` in order to set the label:

```vbnet
Private Sub RefreshTaskPane()
    Select Case Me.HostName
    Case "Excel"
        ...
    Case "Word"
        Dim Pane As ADXWordTaskPanel1 = _
            TryCast( _
                Me.AdxWordTaskPanesCollectionItem1.CurrentTaskPaneInstance, _
                ADXWordTaskPanel1)
        If Pane IsNot Nothing Then
            Pane.Label1.Text = Me.GetInfoString()
        End If
    Case "PowerPoint"
        ...
    Case Else
```
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 Outlook Regions and Advanced Office Task Panes](#).

**Step #15 - Running the COM Add-in**

Choose *Register Add-in Express Project* in the *Build* menu, and restart the host applications. See also [If you use an Express edition of Visual Studio](#).

You can find your add-in in the *COM Add-ins dialog*. See also [Troubleshooting add-in loading](#).
Add-in Express for Office and .net

Your First Microsoft Office COM Add-in

The current cell is A1

There are 1 slides currently selected
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**

The table below provides links to step-by-step instructions for deploying COM add-ins. Find background information in [Deploying Office Extensions](#).

<table>
<thead>
<tr>
<th>How you install the Office extension</th>
<th>A per-user COM add-in</th>
<th>A per-machine COM add-in</th>
</tr>
</thead>
<tbody>
<tr>
<td>A user runs the installer from a CD/DVD, hard disk or local network location</td>
<td><img src="#" alt="Windows Installer" /> <img src="#" alt="ClickOnce" /> ![ClickTwice :)]</td>
<td><img src="#" alt="Windows Installer" /> ![ClickTwice :)]</td>
</tr>
<tr>
<td>A corporate admin uses Group Policy to install your Office extension for a specific group of users in the corporate network; the installation and registration occurs when a user logs on to the domain. For details, please see the following article on our blog: <a href="#">HowTo: Install a COM add-in automatically using Windows Server Group Policy</a></td>
<td><img src="#" alt="Windows Installer" /> ![ClickTwice :)]</td>
<td><img src="#" alt="N/A" /></td>
</tr>
<tr>
<td>A user runs the installer by navigating to a web location or by clicking a link.</td>
<td><img src="#" alt="ClickOnce" /> ![ClickTwice :)]</td>
<td>![ClickTwice :)]</td>
</tr>
</tbody>
</table>

What's next?

Here you can download the project described above, both VB.NET and C# versions; the download link is labeled *Add-in Express for Office and .NET sample projects*.

You may want to check the following sections under **Tips and Notes**:  

- [Recommended Blogs and Videos](#)  
- [Development Tips](#) – typical problems and a **must-read** section [Releasing COM Objects](#);  
- [COM Add-in Tips](#) – solutions for typical problems: the add-in doesn't show the UI elements, etc.;  
- [Command Bars and Controls Tips](#) – if your COM add-in supports pre-Ribbon Office applications;  
- [Debugging and Deploying Tips](#) – if you have never developed an Office extension;  
- [Architecture Tips](#) – if you develop a combination of Office extensions.

Also, in *Add-in Express Components* we describe components that you use to customize the UI of the host application and handling its events.
Your First Microsoft Outlook COM Add-in

The sample project below shows how you create a COM add-in supporting Outlook 2000-2013. The add-in creates two toolbars: for the Outlook Explorer toolbar and Inspector windows, the toolbars show up in the UI of pre-Ribbon Outlook versions. A custom CommandBar button is added to the toolbars as well as to context menu and the Explorer main menu of Outlook 2000-2007. In the Outlook 2007-2013 Inspector, a Ribbon button is added to the Ribbon UI. Also, the add-in creates an advanced task pane supporting Outlook 2000-2013. Finally, a custom property page is added to the Properties dialog of an Outlook folder and a keyboard shortcut is intercepted. The source code of the project – both VB.NET and C# versions – can be downloaded here; the download is labeled Add-in Express for Office and .NET sample projects.

A Bit of Theory

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).

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 advanced task panes in Office 2000-2013. In a COM add-in targeting Outlook, you can add custom option pages to the Tools | Options and Folder Properties dialogs. 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 email when it is being sent; it can cancel saving an Excel workbook; or, it can check if a Word document or selected text meets some requirements.

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. Add-in Express generates a per-user add-in project; your add-in is per-machine if the add-in module has ADXAddinModule.RegisterForAllUsers = True. Registering for all users means creating registry entries in HKLM and that means the user registering a per-machine add-in must have administrative permissions. Accordingly, RegisterForAllUsers = False means writing to HKCU (=for the current user). See Registry Keys.

Before you modify the RegisterForAllUsers property, you must unregister the add-in project on your development PC and make sure that adxlod.dll.manifest is writable.

A standard user may turn a per-user add-in off and on in the COM Add-ins dialog. You use that dialog to check if your add-in is active.
Step #1 - Creating a COM Add-in Project

Make sure that you have administrative permissions before running Visual Studio. Also, if you have Windows Vista, Windows 7, Windows Server 2008 or Windows 8, run Visual Studio via Run as Administrator.

In Visual Studio, open the New Project dialog and navigate to the Extensibility folder.

Choose Add-in Express COM Add-in and click OK.

This starts the COM Add-in project wizard. It allows choosing your programming language and specifying the oldest Office version your add-in needs to support.
Choosing a particular Office version will add corresponding interop assemblies to the project. Later on, you can replace interop assemblies and reference them in your project. If you are in doubt, choose Microsoft Office 2000 as the minimum supported Office version. If you need background information, see Choosing Interop Assemblies.

Choose your programming language and the minimum Office version you need to support and click Next.

The wizard allows creating add-in projects targeting several Office applications.
You can choose one or more Microsoft Office applications in the above window. For every chosen application, the project wizard will do the following:

- copy the corresponding interop assembly to the Interops folder of your project folder,
- add an assembly reference to the project
- add a COM add-in module to the project
- set up the SupportedApp property of the add-in module
- add a property to the add-in module; the property name reflect the name of the chosen Office application; you will use that property to access the object model of the Office application loading your add-in, see Step #11 – Accessing Outlook Objects.

Select which Office applications will be supported by your add-in and click Next.
If you don't know anything about strong names or don't have a special strong name key file, choose Generate new. If you are in doubt, choose Generate new. If, later on, you need to use a specific strong name key file, you will be able to specify its name on the Signing tab of your project properties; you are required to unregister your add-in project before using another strong name.

Choose Generate new or specify an existing .snk file and click Finish.

The project wizard creates and opens a new solution in the IDE.
The solution contains an only project, the COM add-in project. The add-in project contains the AddinModule.vb (or AddinModule.cs) file discussed in the next step.

**Step #2 - Add-in Module**

AddinModule.vb (or AddinModule.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 configure the add-in’s settings 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 the Solution Explorer, right-click AddinModule.vb (or
Add-in Module.cs) and choose View Code in the popup menu.

In the code of the module, pay attention to three points:

- the comment line in the constructor of the module

The comment suggests that you write any initialization code in the AddinInitialize or AddinStartupComplete events of the add-in module, not in the constructor.

- the OutlookApp property

You use this property – it was added by the COM add-in project wizard – as an entry point to the Outlook object model; see Step #11 – Accessing Outlook Objects.

- the CurrentInstance property

This property returns the current instance of the add-in module, a very useful thing when, for example, you need to access the add-in module from e.g. an advanced Outlook region.

**Step #3 - Add-in Module Designer**

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

The add-in module designer view provides access to the following four areas shown in the screenshot below:

- **Add-in Express Toolbox** – (#1 on the screenshot below) it contains commands; clicking a command creates a new Add-in Express component and places it onto the add-in module

- **Add-in module designer** – (#2 on the screenshot below) it is a usual designer

- **In-place designer** – (#3 on the screenshot below) if there’s a visual designer for the currently selected Add-in Express component, then it is shown in this area

- **Help panel** – see #4 in the screenshot below.
To add any Add-in Express component onto the module, you click an appropriate command in the Add-in Express Toolbox (#1 in the screenshot above). Then you select the newly added component and set it up using the visual designer (#3 in the screenshot above) and Properties window. This procedure is demonstrated in the sections below.

Note that the add-in module provides a number of properties. Click the designer surface and specify the name and description of your add-in in the Properties window. Also, pay attention to the RegisterForAllUsers property – it is essential for distinguishing per-user and per-machine COM add-ins; you must unregister the add-in project before modifying this property.
Step #4 - Adding a New Explorer Toolbar

To add a toolbar to the Outlook 2000-2007 Explorer window, click the Add ADX01ExplorerCommandBar command in the Add-in Express Toolbox. This places a new ADX01ExplorerCommandBar component onto the add-in module.

You may use both CommandBar UI Components and Office Ribbon UI Components on the add-in module. When your add-in is loaded in a particular Outlook version, either command bar or ribbon controls will show up. Find additional information in Command Bars in the Ribbon UI.
Select the just added command bar component and, in the Properties window, specify the command bar name in the CommandBarName property.

ADXOlExplorerCommandBar provides context-sensitive properties: they are FolderName, FolderNames, and ItemTypes. The component displays the specified Explorer command bar for every Outlook folder, the name and default item type of which correspond to the values specified in the FolderName, FolderNames, and ItemTypes properties. Note that the FolderName and FolderNames properties require entering the full path to a folder. FolderName also accepts "*" (asterisk), which means "for every folder". This is the only use of the asterisk recognizable in the current version. In other words, Add-in Express does not support template characters in the FolderName(s) properties.

In the screenshot above, you see the properties of the Outlook Explorer command bar that will be shown for every Outlook folder (FolderName = "*" ) the default item type of which is Mail or Task.

The component creates controls on the toolbar the name of which you specify in the CommandBarName property. If the toolbar is missing in Outlook, the component creates it. That is, if you set CommandBarName = "Standard", and add an ADXCommandBarButton to the Controls collection of the ADXOlExplorerCommandBar component, this will create the button on the built-in Standard toolbar, while specifying CommandBarName = "Standard2" will create a new toolbar, Standard2, with the button on it. If the Standard2 toolbar already exists in Outlook, the button will be added to that toolbar. You can download and use our free Built-in Controls Scanner to get the names of all built-in command bars in Outlook 2000-2013.

Step #5 - Adding a New Toolbar Button

Select the command bar component on the add-in module designer and open the in-place designer area. There you'll see the visual designer of the ADXOlExplorerCommandBar component. Use it to add or remove command bar controls.

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. Note that the Style property (not shown in the screenshot) is set to adxMsoButtonIconAndCaption in order to show the icon because command bar buttons do not show icons by default. The screenshot below demonstrates button properties that make the image used in the sample project show up as transparent.

See also Command Bar Control Properties and Events and CommandBar UI Components.
Step #6 - Customizing the Outlook Ribbon UI

To add a new custom tab to the Ribbon UI in Outlook 2007-2013, you use the Add ADXRibbonTab command that places a new ADXRibbonTab component onto 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 2010-2013 Explorer windows as well, set the Ribbons property to OutlookMailRead;OutlookMailCompose;OutlookExplorer.

You use the visual designer of the ADXRibbonTab component to populate a Ribbon tab with Add-in Express components that form the Ribbon interface of your add-in. In this sample, you add a Ribbon tab component 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. Finally, you select the group and add a button. Set the button caption to My
Ribbon Button. Use the ImageList, Image, ImageTransparentColor, and Glyph properties to set the icon for the button. See also Office Ribbon UI Components and Images on Ribbon Controls.

Step #7 - Adding a New Inspector Toolbar

To add a toolbar to Outlook 2000-2003 Inspector windows, use the Add ADXOlInspectorCommandBar command that places a new ADXOlInspectorCommandBar component onto the add-in module.

ADXOlInspectorCommandBar provides the same context-sensitive properties as ADXOlExplorerCommandBar: the properties are FolderName, FolderNames, and ItemTypes. Add-in Express displays the Inspector command bar for every item created or opened in an Outlook folder, the name and default item type of which correspond to the values specified in the FolderName, FolderNames, and ItemTypes properties. Note that the FolderName and FolderNames properties require entering the full path
to a folder. FolderName also accepts "*" (asterisk), which means "for every folder". This is the only use of the asterisk recognizable in the current version; template characters in the FolderName(s) properties are not supported.

In the screenshot above, you see the properties of the Outlook Inspector command bar that will be shown for every Outlook folder (FolderName = "*") the default item type of which is Mail or Task.

If the toolbar name is the same as the name of a built-in command bar of the host application, then the component will create the controls you specify on the built-in toolbar. Otherwise, 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 ADXOlInspectorCommandBar component, this will create the button on the built-in Standard toolbar, while specifying CommandBarName = "Standard2" will create a new toolbar, Standard2, with the button on it. If the Standard2 toolbar already exists in the host application, the button will be added to that toolbar. Use our free Built-in Controls Scanner to get the names of all built-in command bars in any Office 2000-2013 application. For adding a new command bar button onto the toolbar see Step #5 – Adding a New Toolbar Button. See also CommandBar UI Components.
Step #8 - Customizing Main Menu in Outlook 2000-2007

To demonstrate the standard steps required when dealing with built-in CommandBar controls, we will add a custom control to the File / New popup in the Explorer window of Outlook 2000-2007.

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 shown in the screenshot to bind Add-in Express components to the corresponding CommandBar controls built into the host application of your add-in.

Now add an Outlook Inspector Main Menu component onto the add-in module and do the following (all values below are taken from the screenshot above):

- 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; the settings of such a popup are shown in the screenshot below.
- 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.

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 Toolbox provides two Outlook-related main menu components: Explorer Main Menu component and Inspector Main Menu component.

The Ribbon UI replaces the main menu of Inspector windows in Outlook 2007 and all main menus in Outlook 2010 and above. Nevertheless, the main menu as well as all command bars and their controls are still available for the developer, please see Navigating Up and Down the Command Bar System. See also Connecting to Existing CommandBar Controls, CommandBar UI Components and Office Ribbon UI Components.
Step #9 - Customizing Outlook Context Menus

Add-in Express allows customizing commandbar-based context menus of Outlook 2002-2010 via the ADXContextMenu component (Outlook 2000 context menus are not customizable and Outlook 2013 doesn’t support commandbar-based context menus). Click the corresponding command in the Add-in Express Toolbox to add such a component onto the add-in module. Then choose Outlook in the SupportedApp property of the component. Then, in the CommandBarName property, choose the context menu you want to customize. Finally, you add custom controls in the visual designer of the context menu component.

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

Context menus in Outlook 2010 are customizable with both ADXContextMenu (commandbar-based) and ADXRibbonContextMenu (Ribbon-based) components. Note that CommandBar-based context menu items
cannot be positioned in the Ribbon-based context menus of Office 2010: a custom context menu item created with the \texttt{ADXContextMenu} component will always be shown below any built-in or custom context menu items in a Ribbon-based context menu.

Outlook 2013 supports only Ribbon-based context menus.

The \textit{Add ADXRibbonContextMenu} command in the Add-in Express Toolbox places a new \texttt{ADXRibbonContextMenu} component onto the add-in module. Then you set the \texttt{Ribbons} property that supplies context menu names for the \texttt{ContextMenuNames} property of the \texttt{ADXRibbonContextMenu}. Finally, you use the \texttt{ContextMenuNames} property editor to choose the context menu(s) that will display your custom controls specified in the \texttt{Controls} property.

The screenshot below shows the editor window of the \texttt{ContextMenuNames} property. To get the context menu names displayed in the screenshot, you need to set the \texttt{Ribbons} property to \texttt{OutlookMailRead;OutlookMailCompose;OutlookExplorer} as shown in the screenshot above.
See also [CommandBar UI Components][1] and [Office Ribbon UI Components][2].
Step #10 - Adding an Advanced Outlook Region in Outlook 2000-2013

Creating a new Outlook region includes the following steps:

- Use the Add-in Express Toolbox to add an Outlook Forms Manager, `ADXOlFormsManager`, onto the add-in module
- Open the *Add New Item* dialog in Visual Studio to add an Add-in Express Outlook Form, `ADXOlForm`, to the project

- Select the Outlook Forms Manager component and add an item to its *Items* collection
The item of the `ADXOlFormsCollectionItem` type provides properties for showing the form specified in the `FormClassName` property. For this sample project, the properties and their settings are as follows:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>FormClassName</code></td>
<td><code>MyOutlookAddin1.ADXOlForm1</code></td>
<td>The class name of the form, instances of which will be created and shown as specified in other properties.</td>
</tr>
<tr>
<td><code>ExplorerItemTypes</code></td>
<td><code>Mail</code></td>
<td>An instance of the form specified in the <code>FormClassName</code> property will be shown below the list of items in the Outlook Explorer whenever you navigate to a mail folder.</td>
</tr>
<tr>
<td><code>ExplorerLayout</code></td>
<td><code>BottomSubpane</code></td>
<td>An instance of the form specified in the <code>FormClassName</code> property will be shown below the message body whenever you open an email.</td>
</tr>
<tr>
<td><code>InspectorItemTypes</code></td>
<td><code>Mail</code></td>
<td></td>
</tr>
<tr>
<td><code>InspectorLayout</code></td>
<td><code>BottomSubpane</code></td>
<td></td>
</tr>
<tr>
<td><code>AlwaysShowHeader</code></td>
<td><code>True</code></td>
<td>These will show a header containing the form icon and the form caption even if the form is a single form in the given</td>
</tr>
</tbody>
</table>
**CloseButton = True**

The header will contain the *Close* button; when you click it, the form will generate the `OnADXBeforeCloseButtonClick` event (cancellable).

**UseOfficeThemeForBackground = True**

A pre-defined color corresponding to the current Office theme is used for the background of the form specified in the `FormClassName` property.

See also [Introducing Advanced Outlook Form and View Regions](#), [Advanced Outlook Regions](#).
Step #11 - Accessing Outlook Objects

Add the following method to the add-in module:

```vba
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 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 create the following event handlers for the CommandBar and Ribbon buttons added in previous steps:
Private Sub ActionInExplorer(ByVal sender As System.Object) _
 Handles AdxCommandBarButtonClick
 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 AdxCommandBarButtonClick, AdxCommandBarButtonClick6.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 2010-2013,
 End If
 Marshal.ReleaseComObject(context)
End Sub
Step #12 - Handling Outlook Events

The Add-in Express Toolbox provides the Add Events command that creates (and deletes) event components providing application-level events. In this sample, we place the Outlook Events component onto the add-in module.

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

```vba
Private Sub adxOutlookEvents_ExplorerBeforeFolderSwitch(ByVal sender As Object, ByVal e As AddinExpress.MSO.ADXO1ExplorerBeforeFolderSwitchEventArgs) Handles adxOutlookEvents.ExplorerBeforeFolderSwitch
    MsgBox("You are switching to the " + e.NewFolder.Name + " folder")
End Sub
```

If you create a label on the form added in Step #10 – Adding an Advanced Outlook Region in Outlook 2000-2013, you can modify the label in the ADXSelectionChange event of the form:

```vba
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 ItemAdd method of the class:

```vb
Imports System

'Add-in Express Outlook Items Events Class
Public Class OutlookItemsEventsClass1
    Inherits AddinExpress.MSO.ADXOutlookItemsEvents
```
To use this class, you have to add the following declarations and code to the add-in module:

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

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

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

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

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.

See also Outlook Item Events explained and Outlook Items and Folders Events explained.
Step #14 - Adding Property Pages to the Folder Properties Dialog

Outlook allows adding custom pages (tabs) to the Options dialog (the Tools | Options menu) as well as 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).

![Add New Item - MyAddin1](image)

Clicking Add create a descendant of the ADXOlPropertyPage class and adds it to your project. You can customize that page as an ordinary form: add controls and handle their events. To add the property page to the <folder name> Properties dialog box of an Outlook folder(s), follow the steps listed below:

- 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 settings you use to display your page in the Folder Properties dialog for the Inbox.

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. The code below gets the full folder name of the Inbox folder in the AddinStartupComplete event of the add-in module:

```
Private Sub AddinModule_AddinStartupComplete(ByVal sender As Object, ByVal e As System.EventArgs) Handles Me.AddinStartupComplete
    ItemsEvents.ConnectTo(ADXODefaultFolders.olFolderInbox, True)
    Dim ns As Outlook.NameSpace = Me.OutlookApp.Session
    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.
Now add a check box to the property page. The code below handles the CheckedChanged event of the check box as well as the Dirty, Apply, and Load events of the property page:

```csharp
...  
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  
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:

```csharp
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 2010-2013 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:

```vba
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, restart Outlook and find your option page(s), command bars, and controls, Ribbon controls, and custom task panes. See also If you use an Express edition of Visual Studio.
You can also find your add-in in the COM Add-ins dialog. See also Troubleshooting add-in loading.
Hello,

You use Add-in Express according to the following algorithm:
- Create an Add-in Express project
- Add an Add-in Express designer to the project
- Add Add-in Express components to the designer
- Add Add-in Express event classes to your project [optional]
- Add some business logics [required :)]
- Build and register the Add-in Express project
- To deploy your assembly, tune up the Add-in Express Loader based setup project

The subject is:'RE: How do I use Add-in Express for Office and
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

The table below provides links to step-by-step instructions for deploying COM add-ins. Find background information in Deploying Office Extensions.

<table>
<thead>
<tr>
<th>How you install the Office extension</th>
<th>A per-user COM add-in</th>
<th>A per-machine COM add-in</th>
</tr>
</thead>
<tbody>
<tr>
<td>A user runs the installer from a CD/DVD, hard disk or local network location</td>
<td>Windows Installer ClickOnce ClickTwice :)</td>
<td>Windows Installer ClickTwice :)</td>
</tr>
<tr>
<td>A corporate admin uses Group Policy to install your Office extension for a specific group of users in the corporate network; the installation and registration occurs when a user logs on to the domain. For details, please see the following article on our blog: HowTo: Install a COM add-in automatically using Windows Server Group Policy</td>
<td>Windows Installer</td>
<td>N/A</td>
</tr>
<tr>
<td>A user runs the installer by navigating to a web location or by clicking a link</td>
<td>ClickOnce ClickTwice :)</td>
<td>ClickTwice :)</td>
</tr>
</tbody>
</table>

What's next?

Here you can download the project described above, both VB.NET and C# versions; the download link is labeled Add-in Express for Office and .NET sample projects.

You may want to check the following sections under Tips and Notes:

- **Recommended Blogs and Videos**
- **Development Tips** – typical problems and a must-read section Releasing COM Objects;
- **COM Add-in Tips** – solutions for typical problems: the add-in doesn't show the UI elements, etc.
- **Command Bars and Controls Tips** – if your COM add-in supports pre-Ribbon Office applications;
- **Debugging and Deploying Tips** – if you have never developed an Office extension;
- **Architecture Tips** – if you develop a combination of Office extensions.

Also, in Add-in Express Components we describe components that you use to customize the UI of the host application and handling its events.
Your First Excel RTD Server

The sample project below demonstrates how you create an Excel RTD server handling a single topic. The source code of the project – both VB.NET and C# versions – can be downloaded here; the download link is labeled Add-in Express for Office and .NET sample projects.

A Bit of Theory

The RTD Server technology (introduced in Excel 2002) 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 = False means writing to HKCU (=for the current user).

Before you modify the RegisterForAllUsers property, you must unregister the add-in project on your development PC and make sure that adxloader.dll.manifest is writable.
Step #1 - Creating an RTD Server Project

Make sure that you have administrative permissions before running Visual Studio. Also, if you have Windows Vista, Windows 7, Windows Server 2008 or Windows 8, run Visual Studio via Run as Administrator.

In Visual Studio, open the New Project dialog and navigate to the Extensibility folder.

Choose Add-in Express RTD Server and click OK.

This starts the RTD server project wizard.

In the first wizard window, you choose your programming language (see below).
Welcome to the Add-in Express for Microsoft Office and .NET Project Wizard

This will create an RTD server project for Microsoft Excel.

Select a programming language:

- Visual C# Project
- Visual Basic Project
- Visual C++ Project
- Visual J# Project
- Delphi Prism Project

Interop assemblies:

Minimum supported Office version

Microsoft Office XP

Strong name key file:

- Generate new
- Use existing one

Finish
When in the window above, choose *Generate new* or specify an existing .snk file and click *Next*. If you do not know anything about strong names or do not have a special strong name key file, choose *Generate new*. If you are in doubt, choose *Generate new*. If, later on, you need to use a specific strong name key file, you will be able to specify its name on the *Signing* tab of your project properties; you are required to unregister your add-in project before using another strong name.

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

The solution contains an only project, the RTD server project. The project contains 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 `ADXRRTDTopic` components. It is a descendant of the `ADXRRTDServerModule` 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.

In the code of the module, pay attention to the `CurrentInstance` property. It returns the current instance of the RTD module. This is useful for example, when you need to access a method defined in the module from the code of another class.

Step #3 - RTD Server Module 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.

This opens the designer of the RTD server module (see below):
To add components to your class, drag them from the Toolbox and use the Properties window to set their properties. To create methods and events for your class, click here to switch to code view.

The areas shown in the screenshot above are:

- **RTD server module designer** - (#1 on the screenshot above) it is a usual designer
- **Help panel** – see #2 in the screenshot above.

Click the designer surface when you need to set properties of your RTD server in the Properties window. The RegisterForAllUsers property shown in the screenshot above is described in Per-user and Per-machine RTD Servers. The Interval property sets the internal timer that causes Excel to generate the RefreshData event for topics 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 places a new ADXRTDTopic component onto the module. Select the newly added component and, in the Properties window,
enter string values identifying the topic in the String## properties. In this sample, the My Topic string in the String01 property identifies the topic.

It is possible to enter an asterisk (*) in any of the String## properties. When there is no ADXRTDTopic corresponding to the identifying strings entered by the user, Add-in Express creates a new ADXRTDTopic and passes it to the RefreshData event handler of the topic containing an asterisk (*). In that event, you can cast the Sender argument to ADXRTDTopic and get actual strings from its String## properties.

```vbnet
Private Function AdxrtdTopic1_RefreshData(ByVal sender As System.Object) As System.Object
    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, restart Excel, and enter the RTD function to a cell.

See also If you use an Express edition of Visual Studio.

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

The table below provides links to step-by-step instructions for deploying RTD servers. Find background information in Deploying Office Extensions.

<table>
<thead>
<tr>
<th>How you install the Office extension</th>
<th>A per-user RTD server</th>
<th>A per-machine RTD server</th>
</tr>
</thead>
<tbody>
<tr>
<td>A user runs the installer from a CD/DVD, hard disk or local network location</td>
<td>Windows Installer ClickOnce ClickTwice :)</td>
<td>Windows Installer ClickTwice :)</td>
</tr>
<tr>
<td>A corporate admin uses Group Policy to install your Office extension for a specific group of users in the corporate network; the installation and registration occurs when a user logs on to the domain. For details, please see the following article on our blog: HowTo: Install a COM add-in automatically using Windows Server Group Policy</td>
<td>Windows Installer</td>
<td>N/A</td>
</tr>
<tr>
<td>A user runs the installer by navigating to a web location or by clicking a link.</td>
<td>ClickOnce ClickTwice :)</td>
<td>ClickTwice :)</td>
</tr>
</tbody>
</table>

What’s next?

Here you can download the project described above, both VB.NET and C# versions; the download link is labeled Add-in Express for Office and .NET sample projects.

You may want to check the following sections under Tips and Notes:

- Recommended Blogs and Videos
- Development Tips – typical problems;
- RTD Tips – FAQ on RTD Server and a valuable article: Inserting the RTD Function in a User-Friendly Way;
- Architecture Tips – if you develop a combination of Office extensions.

An interesting series of articles describing the creation of a real project from A to Z is available on our blog. The starting point is Building a Real-Time Data server for Excel.
Your First Smart Tag

The sample project below demonstrates how you create a smart tag handling a list of fixed words and providing the words with a sample action. The source code of the project – both VB.NET and C# versions – can be downloaded here; the download link is labeled Add-in Express for Office and .NET sample projects.

A Bit of Theory

Smart Tags were introduced in Word 2002 and Excel 2002. Then they added PowerPoint 2003 to the list of smart tag host applications.

In Office 2010 Microsoft declared smart tags deprecated. Although you can still use the related APIs in projects for Excel, Word, PowerPoint 2010 and 2013, 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.

Below is what was said about the Smart Tag technology in earlier days:

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 email address you type into a Word document or an Excel workbook. When smart tag recognizes the email address, it allows the user to choose one of the actions associated with the text. For email addresses, possible actions are to look up additional contact information or send a new email message to that contact.

Per-user Smart Tags

A smart tag is a per-user thing that requires registering in HKCU. In other words, a smart tag cannot be registered for all users on the machine. Instead, it must be registered for every user separately.
Step #1 - Creating a Smart Tag Library Project

Make sure that you have administrative permissions before running Visual Studio. Also, if you have Windows Vista, Windows 7, Windows Server 2008 or Windows 8, run Visual Studio via Run as Administrator.

In Visual Studio, open the New Project dialog and navigate to the Extensibility folder.

Choose Add-in Express Smart Tag and click OK.

This starts the Smart tag project wizard.

In the first wizard window, you choose your programming language (see below).
Welcome to the Add-in Express for Microsoft Office and .NET Project Wizard

This will create a smart tag project for Microsoft Office.

Select a programming language:

- Visual C# Project
- Visual Basic Project
- Visual C++ Project
- Visual J# Project
- Delphi Prism Project

Interop assemblies:

Minimum supported Office version
Microsoft Office XP

Add-in Express for Office and .net
Your First Smart Tag
In the window above, choose *Generate new* or specify an existing .snk file and click *Next*.

If you do not know anything about strong names or do not have a special strong name key file, choose *Generate new*. If you are in doubt, choose *Generate new*. If, later on, you need to use a specific strong name key file, you will be able to specify its name on the *Signing* tab of your project properties; you are required to unregister your add-in project before using another strong name.

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

In the code of the module, pay attention to the CurrentInstance property. It returns the current instance of the Smart Tag module. This is useful when, for example, you need to access a method defined in the module from the code of another class.

**Step #3 - Smart Tag Module Designer**

The module designer allows setting smart tag properties and adding Smart Tag components to the module.

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

This opens the designer window shown below:
The designer view of the smart tag module provides access to the following two areas shown on the screenshot above:

- **Smart module designer** - (#1 in the screenshot above) it is a usual designer;
- **Help panel** – see #2 in the screenshot above.

Click the designer surface when you need to set properties of the Smart tag module in the *Properties* window.
Step #4 - Adding a New Smart Tag

To add a smart tag to your Smart tag library, you use the Add Smart Tag command (see below) that places a new ADXSmartTag component onto the module.

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 Smart Tag Actions

Now you add smart tag actions to the context menu of your smart tag. 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 (see the screenshot below).

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:

```vbnet
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 the Smart Tag

Choose the Register Add-in Express Project item in the Build menu, restart Word, and enter words recognizable by your smart tag into a document.

See also If you use an Express edition of Visual Studio.

Please remember that Smart tags are declared deprecated in Office 2010-2013. However, 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, you can check if your smart tag is present in the AutoCorrect dialog:

- In Office 2002-2003, 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 2010-2013, see File tab | Options | Add-ins | "Manage" Actions | Go.
Step #7 - Debugging the Smart Tag

To debug your Smart Tag, just specify the host application as the *Start External Program* in the *Project Options* window and press (F5).
## Step #8 - Deploying the Smart Tag

Links to step-by-step instructions for deploying smart tags are given in the table below. Background information is provided in [Deploying Office Extensions](#).

<table>
<thead>
<tr>
<th>How you install the Office extension</th>
<th>A per-user Smart tag</th>
<th>A per-machine Smart tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>A user runs the installer from a CD/DVD, hard disk or local network location</td>
<td>Installs and registers for the user running the installer</td>
<td>Installs and registers for all users on the PC</td>
</tr>
<tr>
<td>A corporate admin uses Group Policy to install your Office extension for a specific group of users in the corporate network; the installation and registration occurs when a user logs on to the domain. For details, please see the following article on our blog: HowTo: Install a COM add-in automatically using Windows Server Group Policy</td>
<td>Windows Installer&lt;br&gt;Cli<strong>kOnce&lt;br&gt;Cli</strong>kTwice :)</td>
<td>N/A</td>
</tr>
<tr>
<td>A user runs the installer by navigating to a web location or by clicking a link.</td>
<td>Cli<strong>kOnce&lt;br&gt;Cli</strong>kTwice :)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

### What's next?

[[Here]](#) you can download the project described above, both VB.NET and C# versions; the download link is labeled Add-in Express for Office and .NET sample projects.

You may want to check [Development Tips](#) where we describe typical misunderstandings and provide useful tips.

If you develop a combination of Office extensions, check [Architecture Tips](#).
Your First Excel Automation Add-in

The sample project below demonstrates how you create an Excel automation add-in providing a sample user-defined function. The source code of the project – both VB.NET and C# versions – can be downloaded here; the download link is labeled Add-in Express for Office and .NET sample projects.

A Bit of Theory

Excel user-defined functions (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 UDF: it should 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 should return a properly dimensioned array of values of the types above. Excel shows the value returned by the function in the cell where the user calls the function.

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

Per-user Excel UDFs

An Excel UDF is a per-user thing that requires registering in HKCU. In other words, a UDF cannot be registered for all users on the machine. Instead, it must be registered for every user separately. See also Registry Keys.
Step #1 - Creating a COM Add-in Project

Make sure that you have administrative permissions before running Visual Studio. Also, if you have Windows Vista, Windows 7, Windows Server 2008 or Windows 8, run Visual Studio via Run as Administrator.

In Visual Studio, open the New Project dialog and navigate to the Extensibility folder.

Choose Add-in Express COM Add-in and click OK.

This starts the COM Add-in project wizard.

The wizard allows choosing your programming language and specifying the oldest Office version your add-in needs to support.
Choosing a particular Office version will add corresponding interop assemblies to the project. Later on, in case you need to support an older or a newer Office version, you will be able to replace interop assemblies and reference them in your project. If you are in doubt, choose Microsoft Office 2002 as the minimum supported Office version (because Excel Automation add-ins are supported in Excel 2002 and higher). If you need background information, see Choosing Interop Assemblies.

Choose your programming language and the minimum Office version that you want to support and click Next.

The wizard allows creating add-in projects targeting several Office applications; you select Excel.
For the settings shown on the screenshot above, the project wizard will do the following:

- copy the corresponding version of Excel interop assembly to the Interops folder of your project folder,
- add an assembly reference to the project
- add a COM add-in module to the project
- set up the SupportedApp property of the add-in module.

Select Excel as the Office application your add-in will support and click Next.
If you don't know anything about strong names or don't have a special strong name key file, choose Generate new. If you are in doubt, choose Generate new. If, later on, you need to use a specific strong name key file, you will be able to specify its name on the Signing tab of your project properties; you are required to unregister your add-in project before using another strong name.

Choose Generate new or specify an existing .snk file and click Next.

The project wizard creates and opens a new solution in the IDE.
The solution contains an only project, the COM add-in project.
Step #2 - Adding a COM Excel Add-in Module

Open the Add New Item dialog for the COM add-in project and navigate to Excel below Add-in Express Items. 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.

Choose COM Excel Add-in Module and click Add. This adds the ExcelAddinModule1.vb (or ExcelAddinModule1.cs) file to the COM add-in project.
Step #3 - Writing a User-Defined Function

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

Add a new public function to the class and write the code below:

```vbnet
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, restart Excel, and check if your add-in works.

See also If you use an Express edition of Visual Studio.

You can find your Excel add-in in the Add-ins dialog, see Excel Add-ins dialog.
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**

The table below provides links to step-by-step instructions for deploying Excel Automation add-ins. Find background information in Deploying Office Extensions.

<table>
<thead>
<tr>
<th>How you install the Office extension</th>
<th>A per-user Excel UDF</th>
<th>A per-machine Excel UDF</th>
</tr>
</thead>
<tbody>
<tr>
<td>A user runs the installer from a CD/DVD, hard disk or local network location</td>
<td>Windows Installer ClickOnce ClickTwice :)</td>
<td>N/A</td>
</tr>
<tr>
<td>A corporate admin uses Group Policy to install your Office extension for a specific group of users in the corporate network; the installation and registration occurs when a user logs on to the domain. For details, please see the following article on our blog: HowTo: Install a COM add-in automatically using Windows Server Group Policy</td>
<td>Windows Installer</td>
<td>N/A</td>
</tr>
<tr>
<td>A user runs the installer by navigating to a web location or by clicking a link.</td>
<td>ClickOnce ClickTwice :)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

**What's next?**

Here you can download the project described above, both VB.NET and C# versions; the download link is labeled Add-in Express for Office and .NET sample projects.

You may want to check the following sections under Tips and Notes:

- **Recommended Blogs and Videos**
- **Development Tips** – typical misunderstandings, useful tips and a must-read section Releasing COM Objects;
- **Excel UDF Tips** – many useful articles on developing Excel user-defined functions including What Excel UDF Type to Choose?

If you develop a combination of Excel extensions, check Architecture Tips and HowTo: Create a COM add-in, XLL UDF and RTD server in one assembly.
Your First XLL Add-in

The sample project below demonstrates how you create an XLL add-in providing a sample user-defined function allocated to a custom function category. The source code of the project – both VB.NET and C# versions – can be downloaded here; the download link is labeled Add-in Express for Office and .NET sample projects.

A Bit of Theory

Excel user-defined functions (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 UDF: it should 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 should return a properly dimensioned array of values of the types above. Excel shows the value returned by the function in the cell where the user calls the function.

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

Per-user Excel UDFs

An Excel UDF is a per-user thing that requires registering in HKCU. In other words, a UDF cannot be registered for all users on the machine. Instead, it must be registered for every user separately. See also Registry Keys.
Step #1 - Creating an XLL Add-in Project

Make sure that you have administrative permissions before running Visual Studio. Also, if you have Windows Vista, Windows 7, Windows Server 2008 or Windows 8, run Visual Studio via Run as Administrator.

In Visual Studio, open the New Project dialog and navigate to the Extensibility folder.

Choose Add-in Express XLL Add-in and click OK.

This starts the XLL Add-in project wizard.

In the first wizard window, you choose your programming language and specify the oldest Excel version your add-in needs to support (see below).
When in the window above, choose Generate new or specify an existing .snk file and click Next. If you don't know anything about strong names or don't have a special strong name key file, choose Generate new. If you are in doubt, choose Generate new. If, later on, you need to use a specific strong name key file, you will be able to specify its name on the Signing tab of your project properties; you are required to unregister your add-in project before using another strong name.

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

The solution contains an only project, the XLL add-in project. The project contains the XLLModule.vb (or XLLModule.cs) file discussed in the next step.
Step #2 - XLL Module

The XLLModule.vb (or XLLModule.cs) file is the core part of the XLL add-in project. The XLL module allows creating and configuring custom user-defined functions (UDF). To review the code, in Solution Explorer, right-click the file and choose View Code in the context menu.

In the code of the module, pay attention to three points:

- the XLLContainer class

That class is the container class for your UDFs; all of them must be declared public static (Public Shared in VB.NET). An UDF must return a string, double or integer. Please see the next step for the use of this class.

- the ExcelApp property

This property was added by the COM add-in project wizard. You use it as an entry point to the Excel object model if this is required in your add-in.

- the CurrentInstance property

This property returns the current instance of the XLL module, a very useful thing when, for example, you need to access a method defined in the module from the code of another class.

Step #3 - Creating a User-Defined Function

Add a new public Shared (static in C#) function to the XLLContainer class.

```csharp
Public Shared Function MyFunction(ByVal arg As Object) As Object
    If TypeOf arg Is Double Then
        Return rnd.NextDouble()
    Else
        If _Module.IsInFunctionWizard Then
            Return "The parameter must be numeric!"
        Else
            Return AddinExpress.MSO.ADXExcelError.xlErrorNum
        End If
    End If
End Function
```
The method above demonstrates the use of the `IsInFunctionWizard` property; it returns `True` if your UDF is called from the Insert Function wizard. In addition, it demonstrates how to return an error such as `#NUM!`

**Step #4 - Configuring UDFs**

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

In **Solution Explorer**, right-click `XLLModule.vb` (or `XLLModule.cs`) and choose **View Designer** in the popup menu.
The XLL module designer provides the three areas shown in the screenshot above. They are:

- **XLL module designer** - (#1 in the screenshot above) it is a usual designer;
- **In-place designer** - (#2 in the screenshot above) if there's a visual designer for the currently selected Add-in Express component, then it is shown in this area;
- **Help panel** – see #3 in the screenshot above.

Specify the add-in name in the *Properties* window. Right-click the XLL module designer and choose *Add Excel Function Category* in the context menu.

This places a new *Excel Function Category* component onto the XLL module.
You use the toolbar provided by the in-place designer to select, move, add or remove components. In this sample, you specify properties of the Excel Function Category component as demonstrated in the screenshot above and add a new function descriptor to the category component as shown below:
In the function descriptor, you set the FunctionName property, which provides a combo box that allows choosing a function from the list of functions defined in the XLLContainer class; select the function you created in Step #3. Other properties are:

- *IsHidden* allows to hide the function from the UI;
- *IsThreadSafe* – you can mark your function as safe for multi-threaded recalculations (Excel 2007+);
- *IsVolatile = True* means that your function will be recalculated whenever calculation occurs in any cell(s) of the worksheet; a nonvolatile function is recalculated only when the input variables change;
- *UnicodeName* – allows specifying a language-specific function name (if the Localizable property of the XLL module is set to True).
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).

Other properties are described below:

- `AcceptAsArray = True` means that your code will receive an array of parameters when the user passes a range to your UDF; otherwise, an instance of `ADXExcelRef` will be passed to your code.
- `UnicodeName` allows specifying a language-specific name for the argument (if the `Localizable` property of the XLL module is set to `True`).

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 the XLL Add-in

Choose Register Add-in Express Project in the Build menu, restart Excel, and check if your add-in works.

See also If you use an Express edition of Visual Studio.

You can find your Excel add-in in the Add-ins dialog, see Excel Add-ins dialog.

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

![Insert Function dialog]

![Function Arguments dialog]
Step #6 - Debugging the XLL 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

The table below provides links to step-by-step instructions for deploying XLLs. Find background information in Deploying Office Extensions.

<table>
<thead>
<tr>
<th>How you install the Office extension</th>
<th>A per-user XLL add-in installs and registers for the user running the installer</th>
<th>A per-machine XLL add-in installs and registers for all users on the PC</th>
</tr>
</thead>
<tbody>
<tr>
<td>A user runs the installer from a CD/DVD, hard disk or local network location</td>
<td>Windows Installer ClickOnce ClickTwice :)</td>
<td>N/A</td>
</tr>
<tr>
<td>A corporate admin uses Group Policy to install your Office extension for a specific group of users in the corporate network; the installation and registration occurs when a user logs on to the domain. For details, please see the following article on our blog: HowTo: Install a COM add-in automatically using Windows Server Group Policy</td>
<td>Windows Installer</td>
<td>N/A</td>
</tr>
<tr>
<td>A user runs the installer by navigating to a web location or by clicking a link.</td>
<td>ClickOnce ClickTwice :)</td>
<td>N/A</td>
</tr>
</tbody>
</table>

What’s next?

Here you can download the project described above, both VB.NET and C# versions; the download link is labeled Add-in Express for Office and .NET sample projects.

- Recommended Blogs and Videos
- Development Tips – typical misunderstandings, useful tips and a must-read section Releasing COM Objects;
- Excel UDF Tips – many useful articles on developing Excel user-defined functions including What Excel UDF Type to Choose?

If you develop a combination of Excel extensions, check Architecture Tips and HowTo: Create a COM add-in, XLL UDF and RTD server in one assembly.
Add-in Express Components

The components Add-in Express provides are described in these sections:

- **Add-in Express Modules** – every Add-in Express based project is built around a module of the corresponding type;
- **Office Ribbon UI Components** – you customize the Ribbon UI with components described in this section. Remember, the Ribbon was introduced in several Office 2007 applications and Outlook 2007 provides the Ribbon for the Inspector window only; the Explorer window in Outlook 2007 uses the CommandBar UI;
- **CommandBar UI Components** – conventional toolbars live in Office 2000-2003 and some Office 2007 applications (including the Explorer window of Outlook 2007); you use these components together with the Ribbon components to create an add-in supporting all Office versions;
- **Custom Task Panes in Office 2007-2013** – in Office 2007 Microsoft allowed you to add custom panes to Outlook, Excel, Word and PowerPoint; later on, they added Project 2010 to the list of applications supporting custom task panes;
- **Advanced Outlook Regions and Advanced Office Task Panes** – Add-in Express provides the technology allowing creating custom regions in Outlook 2000-2013 and panes in Excel, Word and PowerPoint 2000-2013;
- **Events** – these are application-level events of all Office applications, events of Outlook items and collections, as well as keyboard shortcuts;
- **Outlook UI Components** – Outlook shortcut links and Outlook property pages (shown, say, in properties of a folder;
- **Custom Toolbar Controls** – using .NET controls on command bars (not available for the Ribbon UI).
Add-in Express Modules

An Add-in Express based project is built around the module that represents the Office extension you are creating. The module implements the interfaces required for your add-in to be loaded by the corresponding Office application. Also, it contains the code executed when the add-in is being registered and unregistered. In addition, the module provides a design surface, you use it to store the components required by the add-in's functionality.

Specific types of Add-in Express modules are described in these articles:

- COM Add-in Module: Your First Microsoft Office COM Add-in and Your First Microsoft Outlook COM Add-in
- RTD Server Module: Your First Excel RTD Server
- Smart Tag Module: Your First Smart Tag
- COM Excel Add-in Module: Your First Excel Automation Add-in
- XLL Add-in Module: Your First XLL Add-in
- COM Add-in Additional Module: How to Develop the Modular Architecture of your COM and XLL Add-in
- XLL Add-in Additional Module: How to Develop the Modular Architecture of your COM and XLL Add-in
- ClickOnce Module: Customizing ClickOnce installations
- ClickTwice Module: Customizing ClickTwice installations
Office Ribbon UI Components

Starting from version 2007 Office provides the Ribbon user interface. Microsoft states that the interface makes it easier and quicker for users to achieve the wanted results. You extend this interface by using the XML markup that the COM add-in returns to the host application through an appropriate interface when your add-in is loaded into the host 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 (also known as Backstage View). When the add-in is being loaded in Office 2010 or 2013, ADXRibbonOfficeMenu maps your controls to the File tab unless you have an ADXBackStageView component in your add-in; in this case, all the controls you add to ADXRibbonOfficeMenu are ignored.

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 Outlook COM Add-in, Your First Microsoft Office 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 (in a pre-Ribbon Office application, the very first event is `OnAddinInitialize`). 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 2010-2013 only, `AddinExpress.MSO.IRibbonUI` allows 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 check box named "Show add-in user interface errors", see Get Informed about Errors in Ribbon markup and here.

Referring to Built-in Ribbon Controls

All built-in Ribbon controls are identified by their IDs. While the ID of a command bar control is an integer, the ID of a built-in Ribbon control is a string. IDs of built-in Ribbon controls can be downloaded on the Microsoft web site, for Office 2007, see here; for Office 2010, see this page; for Office 2013, see here. 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 (see the screenshot below). You may find these files useful when dealing with changes in the Office 2013 Ribbon UI. Find more details about using them here.
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 ID of a built-in Ribbon control to be intercepted in the *IdMso* property of the component. To get such an ID, see Referring to Built-in Ribbon Controls.

**Disabling Built-in Ribbon Controls**

The *Ribbon Command* component allows disabling built-in Ribbon controls such as buttons, check boxes, menus, groups, etc. To achieve this you need to specify the *IdMso* of the corresponding Ribbon control (see Referring to Built-in Ribbon Controls), set an appropriate value to the *ActionTarget* property, and specify *Enabled=false*. Below are two examples showing how you use the *Ribbon Command* component to prevent the user from 1) copying the selected text and 2) changing the font size of the selected text.

**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 \texttt{InsertBeforeIdMso} and \texttt{InsertAfterIdMso} properties allow positioning the control among built-in Ribbon controls (see also Referring to Built-in Ribbon Controls).

Images on Ribbon Controls

Most Ribbon controls in Office require 32×32 or 16×16 icons. A Ribbon gallery allows using 16×16, 32×32, 48×48, or 64×64 icons. Supported formats are BMP, PNG and ICO, any color depth.

You specify the icon using either the \texttt{ImageList}, \texttt{Image}, \texttt{ImageTransparentColor} properties or the \texttt{Glyph} property that the corresponding Ribbon component provides. Note that \texttt{Glyph} allows bypassing a bug in the \texttt{ImageList} component: \texttt{ImageList} cuts the alpha channel off.

Creating Ribbon Controls at Run Time

You cannot create Ribbon controls at run time because Ribbon is a static thing from birth; but see How Ribbon Controls Are Created? The only control providing any dynamism is \textit{Dynamic Menu}; if the \texttt{ADXRibbonMenu.Dynamic} property is set to \texttt{True} at design-time, the component will generate the \texttt{OnCreate} event allowing creating menu items at run time (see sample code below). For other control types, you can only imitate that dynamism by setting the \texttt{Visible} property of a Ribbon control.

For instance, you may need to display a custom control positioned before or after some other built-in control. To achieve this you create two controls, specify the \texttt{IdMso} of the built-in control in the \texttt{BeforeIdMso} and \texttt{AfterIdMso} properties of the custom controls. At run time, you just change the visibility of the custom controls so that only one of them is visible.

The sample code below demonstrates creating a custom control in the \texttt{OnCreate} event of an \texttt{ADXRibbonMenu} component. For the sample to work, the \texttt{Dynamic} property of the component must be set to \texttt{True}. Note that the \texttt{OnCreate} event occurs whenever you open a dynamic menu.

```csharp
//C#
private ADXRibbonButton newButton;
private void adxRibbonMenu1_OnCreate(object sender, ADXRibbonCreateMenuEventArgs e) {
  e.Clear(); // this removes existing controls

  newButton = new ADXRibbonButton();
  newButton.Id = "newButton";
  newButton.Ribbons = ADXRibbons.msrExcelWorkbook; // specify a required Ribbon(s) here
  newButton.OnClick += new ADXRibbonOnAction_EventHandler(button_OnClick);
  e.AddControl(newButton);
}
```
private void button_OnClick(object sender, IRibbonControl control, bool pressed)
{
    // use a control's ID to identify it
    System.Windows.Forms.MessageBox.Show(control.Id);
}

'VB.NET
Private newButton As ADXRibbonButton
Private Sub adxRibbonMenu1_OnCreate(sender As Object,
    e As ADXRibbonCreateMenuEventArgs) Handles AdxRibbonMenu1.OnCreate
    e.Clear() ' this removes existing controls
    newButton = New ADXRibbonButton()
newButton.Caption = "Created at " & System.DateTime.Now.ToLongTimeString()
newButton.Id = "newButton"
newButton.Ribbon = ADXRibbons.msrExcelWorkbook ' specify a required Ribbon(s) here
AddHandler newButton.OnClick,
    New ADXRibbonOnAction_EventHandler(AddressOf button_OnClick)
e.AddControl(newButton)
End Sub

Private Sub button_OnClick(sender As Object, control As IRibbonControl,
    pressed As Boolean)
    ' use a control's ID to identify it
    System.Windows.Forms.MessageBox.Show(control.Id)
End Sub

Another example creates a dynamic sub-menu in a menu, which is dynamic as well. Here we also demonstrate using e.Contains() to check if a control exists on the menu.

C#
private void adxRibbonMenu1_OnCreate(object sender, ADXRibbonCreateMenuEventArgs e) {
    AddinExpress.MSO.ADXRibbonButton button = null;
    if (!e.Contains("ID_buttonAddItem")) {
        button = new AddinExpress.MSO.ADXRibbonButton();
        button.Caption = "Add an item to submenu";
        button.Id = "ID_buttonAddItem";
        button.Ribbon = AddinExpress.MSO.ADXRibbons.msrExcelWorkbook;
        button.OnClick += buttonAddItem_OnClick;
        e.AddControl(button);
    }

    if (!e.Contains("ID_buttonClearMenu")) {
        button = new AddinExpress.MSO.ADXRibbonButton();
button.Caption = "Clear submenu";
button.Id = "ID_buttonClearMenu";
button.Ribbons = AddinExpress.MSO.ADDRIBBONS.msExcelWorkbook;
button.OnClick += buttonClearMenu_OnClick;
e.AddControl(button);
}

if (!e.Contains("ID_separator")) {
    AddinExpress.MSO.ADDRIBBONSEPARATOR separator =
        new AddinExpress.MSO.ADDRIBBONSEPARATOR();
    separator.Id = "ID_separator";
e.AddControl(separator);
}

if (!e.Contains("ID_submenu")) {
    AddinExpress.MSO.ADDRIBBONMENU subMenu =
        new AddinExpress.MSO.ADDRIBBONMENU();
    subMenu.Caption = "subMenu";
    subMenu.Dynamic = true;
    subMenu.Id = "ID_submenu";
    subMenu.Ribbons = AddinExpress.MSO.ADDRIBBONS.msExcelWorkbook;
    subMenu.OnCreate += subMenu_OnCreate;
e.AddControl(subMenu);
}
}

System.Collections.Specialized.StringCollection items =
    new System.Collections.Specialized.StringCollection();

private void buttonAddItem_OnClick(object sender, IRibbonControl control, bool pressed) {
    items.Add(System.DateTime.Now.ToLongTimeString());
}

private void buttonClearMenu_OnClick(object sender, IRibbonControl control, bool pressed) {
    items.Clear();
}

private void subMenu_OnCreate(object sender, ADDRIBBONCREATEMENUEventArgs e) {
    int counter = 0;
    if (items.Count == 0) e.Clear();
    foreach (string str in items) {
        string strId = "ID_" + counter.ToString();
        if (!e.Contains(strId)) {
            AddinExpress.MSO.ADDRIBBONBUTTON button =
                new AddinExpress.MSO.ADDRIBBONBUTTON();
            button.Caption = str;
            button.Id = strId;
            button.Tag = counter;
        }
    }
```csharp
private void buttonItem_OnClick(object sender, IRibbonControl control, bool pressed) {
    AddinExpress.MSO.ADXRibbonButton button = sender as AddinExpress.MSO.ADXRibbonButton;
    int index = button.Tag;
    MessageBox.Show("You've clicked the item '" + items[index] + "'");
}

private void buttonItem_OnClick(object sender, IRibbonControl control, bool pressed) {  
    AddinExpress.MSO.ADXRibbonButton button = sender as AddinExpress.MSO.ADXRibbonButton;
    int index = button.Tag;
    MessageBox.Show("You've clicked the item '" + items[index] + "'");
}
```

```vbnet
Private Sub AdxRibbonMenu1_OnCreate(sender As System.Object, 
    e As ADXRibbonCreateMenuEventArgs) Handles AdxRibbonMenu1.OnCreate
    Dim button As AddinExpress.MSO.ADXRibbonButton

    If Not e.Contains("ID_buttonAddItem") Then
        button = New AddinExpress.MSO.ADXRibbonButton()
        button.Caption = "Add an item to submenu"
        button.Id = "ID_buttonAddItem"
        button.Ribbons = AddinExpress.MSO.ADXRibbons.msrExcelWorkbook
        AddHandler button.OnClick, AddressOf buttonAddItem_OnClick
        e.AddControl(button)
    End If

    If Not e.Contains("ID_buttonClearMenu") Then
        button = New AddinExpress.MSO.ADXRibbonButton()
        button.Caption = "Clear submenu"
        button.Id = "ID_buttonClearMenu"
        button.Ribbons = AddinExpress.MSO.ADXRibbons.msrExcelWorkbook
        AddHandler button.OnClick, AddressOf buttonClearMenu_OnClick
        e.AddControl(button)
    End If

    If Not e.Contains("ID_separator") Then
        Dim separator As AddinExpress.MSO.ADXRibbonSeparator = 
            New AddinExpress.MSO.ADXRibbonSeparator()
        separator.Id = "ID_separator"
        e.AddControl(separator)
    End If

    If Not e.Contains("ID_subMenu") Then
```
Dim submenu As AddinExpress.MSO.ADXRibbonMenu = New AddinExpress.MSO.ADXRibbonMenu()
submenu.Caption = "submenu"
submenu.Dynamic = True
submenu.Id = "ID_submenu"
submenu.Ribbons = AddinExpress.MSO.ADXRibbons.msrExcelWorkbook
AddHandler submenu.OnCreate, AddressOf submenu_OnCreate
e.AddControl(submenu)
End If
End Sub


Private Sub buttonAddItem_OnClick(sender As System.Object, control As AddinExpress.MSO.IRibbonControl, pressed As System.Boolean)
items.Add(System.DateTime.Now.ToString())
End Sub

Private Sub buttonClearMenu_OnClick(sender As System.Object, control As AddinExpress.MSO.IRibbonControl, pressed As System.Boolean)
items.Clear()
End Sub

Private Sub submenu_OnCreate(sender As System.Object, e As AddinExpress.MSO.ADXRibbonCreateMenuEventArgs)
Dim counter As Integer = 0
If items.Count = 0 Then
e.Clear()
End If
For Each str As String In items
Dim strId As String = "ID_" + counter.ToString()
If Not e.Contains(strId) Then
Dim button As AddinExpress.MSO.ADXRibbonButton = New AddinExpress.MSO.ADXRibbonButton()
button.Caption = str
button.Id = strId
button.Tag = counter
button.Ribbons = AddinExpress.MSO.ADXRibbons.msrExcelWorkbook
AddHandler button.OnClick, AddressOf buttonItem_OnClick
e.AddControl(button)
End If
counter += 1
Next
End Sub

Private Sub buttonItem_OnClick(sender As System.Object,
Add-in Express for Office and .net

Office Ribbon UI Components

Updating Ribbon Controls at Run Time

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 whenever it requests that property. This mechanism is 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 property value whenever the Ribbon invokes a corresponding callback function.

```vbnet
adxRibbonButton1.Caption = "New Caption";
```

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 PropertyChanging 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: Caption, Visible, Enabled, Tooltip, etc. This event allows you to learn the current context (see Determining a Ribbon Control's Context), the requested property and its current value. You can change that value as required by the business logic of your add-in.

```vbnet
'VB.NET
Private Sub AdxRibbonButton1_PropertyChanging(sender As System.Object, _
    e As AddinExpress.MSO.ADXRibbonPropertyChangingEventArgs) _
Handles AdxRibbonButton1.PropertyChanging
Select Case e.PropertyType
    Case AddinExpress.MSO.ADXRibbonControlPropertyType.Caption
        If condition Then
            e.Value = "Some caption"
        Else
            e.Value = "Some other caption"
        End If
    Case AddinExpress.MSO.ADXRibbonControlPropertyType.Enabled
        If condition Then
            e.Value = True
        Else
            e.Value = False
        End If
```
Determining a Ribbon Control's Context

A Ribbon control is shown in a certain context. For the developer, the context is either `null` (*Nothing* in VB.NET) or a COM object that you might need to release after use (according to the rule given in Releasing COM Objects). You retrieve and release the context object in these ways:

- in action events such as `Click` and `Change`, you use the `IRibbonControl` parameter to retrieve `IRibbonControl.Context`; you **must release** this COM object after use;
- in the `PropertyChanging` event (see Updating Ribbon Controls at Run Time), the context is supplied in the `e.Context` parameter; since the COM object is supplied in the parameters of the event handler, you **must not release** it.

For a Ribbon control shown on a Ribbon tab, the context represents the window in which the Ribbon control is shown: `Excel.Window`, `Word.Window`, `PowerPoint.DocumentWindow`, `Outlook.Inspector`, `Outlook.Explorer`, etc. For a Ribbon control shown in a Ribbon context menu the context object may not be a window e.g. `Outlook.Selection`, `Outlook.AttachmentSelection`, etc. When debugging the add-in we recommend that you find the actual type name of the context object by using...
private void OnClick(object sender, IRibbonControl control, bool pressed) {
    object context = control.Context; if (context == null) return;
    if (context is Outlook.Explorer) {
        Outlook.Explorer explorer = context as Outlook.Explorer;
        Outlook.Selection selection = null;
        try {
            selection = explorer.Selection;
        } catch (Exception ex) { }
        if (selection != null) {
            if (selection.Count > 0) {
                object item = selection[1];
                if (item is Outlook.MailItem) {
                    Outlook.MailItem mail = item as Outlook.MailItem;
                    // process mail
                }
                Marshal.ReleaseComObject(item); item = null;
            }
            Marshal.ReleaseComObject(selection); selection = null;
        }
    } else if (context is Outlook.Inspector) {
        Outlook.Inspector inspector = context as Outlook.Inspector;
        object item = inspector.CurrentItem;
        if (item is Outlook.MailItem) {
            Outlook.MailItem mail = item as Outlook.MailItem;
            // process mail
        }
        Marshal.ReleaseComObject(item); item = null;
    } else if (context is Outlook.AttachmentSelection) {
        Outlook.AttachmentSelection selectedAttachments = context as Outlook.AttachmentSelection;
        // process selectedAttachments
    } else if (context is Excel.Window) {
        Excel.Window window = context as Excel.Window;
        Excel.Workbook workbook = window.Parent as Excel.Workbook;
        // process workbook
        Marshal.ReleaseComObject(workbook); workbook = null;
    } else if (context is Word.Window) {
        Word.Window window = context as Word.Window;
        // process document
        Marshal.ReleaseComObject(document); document = null;
    }
    Marshal.ReleaseComObject(context); context = null;
}
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 with a button (private again), the resulting XML markup, which you can get in the OnRibbonBeforeLoad event of the add-in module, looks as follows:

```xml
<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>
CommandBar UI Components

Command bar is a common term for traditional toolbars, menus, and context menus. 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, and 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 Command Bar 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.
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 Add-in Express Toolbox. It places a new ADXCommandBar component onto 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 custom CommandBar in the Add-ins tab in a Ribbon-enabled Office version, 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 this 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 Express Toolbox, you add an ADXMainMenu, which is intended for customizing the main menu in an Office application, which you specify in the SupportedApp property of the component.

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 popup control. For instance, the ID of the File menu item (which is a popup control, in fact) in all Office applications is 30002. Find more details about IDs of command bar controls in Connecting to Existing CommandBar Controls.
In main applications of Office 2007, they replaced the command bar 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 2010-2013. 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 2010-2013, 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 Express Toolbox adds a command bar control to any context menu available in all Office 2000-2013 applications except for Outlook 2000 and Outlook 2013. The component allows connecting to a single context menu of a single host application; to customize several context menus, add an appropriate number of context menu components. 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 #9 – Customizing Outlook Context Menus and Step #8 – Customizing Context Menus.

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.

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 check box 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 2000-2013 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 creates it on the command bar.

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

- Place a new toolbar component onto 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.


**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 `IDTExtensibility2` 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 corresponding command bar component. If you set this property 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 2010-2013 provides support for both commandbar-based and Ribbon-based context menus; see Step #9 – Customizing Outlook Context Menus and Step #8 – Customizing Context Menus.

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 drop downs.

Nevertheless, this list is extremely short. Add-in Express allows extending this list with any .NET control (see Custom Toolbar Controls). 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).
Custom Task Panes in Office 2007-2013

To allow further customization of Office applications, Microsoft 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 `DockPositionStateChanged` and `VisibleStateChanged` events of the task pane item. Use the `OfficeColorSchemeChanged` event and the `OfficeColorScheme` property to get the current Office color scheme.

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** – 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.

```vbscript
Private Sub RefreshTaskPaneInOutlookWindow(ByVal ExplorerOrInspector As Object)
    If Me.HostMajorVersion >= 12 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 UserControl1. The GetSubject method is shown below.

```vbscript
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
            End Try
        If Not selection Is Nothing Then Marshal.ReleaseComObject(selection)
    ElseIf TypeOf ExplorerOrInspector Is Outlook.Inspector Then
        Try
            mailItem = CType(ExplorerOrInspector, Outlook.Inspector).CurrentItem
        Catch
            End Try
    End If
    Dim subject As String = ""
    If mailItem IsNot Nothing Then
        subject = "The subject is: " + mailItem.Subject
        Marshal.ReleaseComObject(mailItem)
    End If
    Return subject
```
The code of the \textit{GetSubject} method emphasizes the following:

- The \texttt{ExplorerOrInspector} parameter was originally obtained through parameters of Add-in Express event handlers. That is why we do not release it (see \texttt{Releasing COM Objects}).
- The \texttt{selection} and \texttt{mailItem} COM objects were created "manually" so they must be released.
- All Outlook versions fire an exception when you try to obtain the \texttt{Selection} object for a top-level folder, such as \textit{Personal Folders}.

Below is another sample that demonstrates how the same things can be done in Excel or Word.

\begin{verbatim}
Imports AddinExpress.MSO
...
Private Sub RefreshTaskPane()
    If Me.HostMajorVersion >= 12 Then
        Dim Window As Object = Me.HostApplication.ActiveWindow
        If Not Window Is Nothing Then
            RefreshTaskPaneInstance(AdxTaskPane1.Item(Window))
            Marshal.ReleaseComObject(Window)
        End If
    End If
End Sub

Private Sub RefreshTaskPaneInstance(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
\end{verbatim}

The \texttt{InfoString} property mentioned above just updates the text of the label located on the \texttt{UserControl}. Please pay attention to \texttt{Releasing COM Objects} in this code.
Advanced Outlook Regions and Advanced Office Task Panes

Add-in Express allows COM add-ins to show Advanced Form and View Regions in Outlook and Advanced Task Panes in Excel, Word, and PowerPoint; versions 2000-2013 are supported.

Introducing Advanced Task Panes in Word, Excel and PowerPoint

In Add-in Express terms, an advanced Office task pane is a sub-pane, or a dock, of the main Excel, Word or PowerPoint window that may host native .NET forms. The screenshot below shows a sample task pane embedded into all available Excel docks.
Introducing Advanced Outlook Form and View Regions

In Add-in Express terms, an advanced Outlook region is a sub-pane, or a dock, of Outlook windows that may host native .NET forms. There are two types of the advanced regions – Outlook view regions (sub-panes on the Outlook Explorer window) and Outlook form regions (sub-panes of the Outlook Inspector window).

Outlook view regions are specified in the ExplorerLayout property of the item (= ADXOlFormsCollectionItem). Outlook form regions are specified in the InspectorLayout property of the item. That is, one ADXOlFormsCollectionItem may show your form in a view and form region. 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 view regions:

- Four regions around the list of mails, tasks, contacts etc. The region names are LeftSubpane, TopSubpane, RightSubpane, BottomSubpane (see the screenshot below). A restriction: these regions are not available for Calendar folders in Outlook 2010-2013.
- One region below the Navigation Pane – BottomNavigationPane (see the screenshot below)
- One region below the To-Do Bar – BottomTodoBar (see the screenshot below). A restriction: this region is not available in Outlook 2013.
- One region below the Outlook Bar (Outlook 2000 and 2002 only) – BottomOutlookBar
- Four regions around the Explorer window (Outlook 2007-2013 only) – **DockLeft, DockTop, DockRight, DockBottom** (see the screenshot below). The restrictions for these regions are:

1. Docked regions are not available for pre-2007 versions of Outlook
2. The Hidden region state is not supported in docked layouts
3. Docked panes have limitations on the minimum height or width
• Four regions around the Reading Pane – `LeftReadingPane`, `TopReadingPane`, `RightReadingPane`, `BottomReadingPane` (see the screenshot below).

![Screenshot of regions around the Reading Pane](image1.png)

• 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`).

![Screenshot of WebViewPane](image2.png)
• The *FolderView* region (see two screenshots below). Unlike *WebViewPane*, it allows the user to switch between the original Outlook view and your form. **A restriction**: this region is not available for Calendar folders in Outlook 2010-2013.

• The *ReadingPane* region (see two screenshots above).
And here is the list of Outlook form regions:

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

- The InspectorRegion region (see two screenshots below) allows switching between your form and the Outlook inspector pane.
The *CompleteReplacement* inspector region shown in the screenshot below is similar to the *InspectorRegion* with two significant differences: a) it doesn’t show the header and in this way, it doesn’t allow switching between your form and the Outlook inspector pane and b) it is activated automatically.
The UI Mechanics

An Absolute Must-Know

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

- there are application-specific `<Manager>` components such as `ADXOlFormsManager` or `ADXExcelTaskPanesManager`; every `<Manager>` component provides the `Items` collection; each `<Item>` from the collection binds a `<Form>`, which is an application-specific descendant of `System.Windows.Forms.Form` such as `ADXOlForm` or `ADXExcelTaskPane`, to the visualization (Excel, Word, PowerPoint and Outlook) 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 sub-pane of the host window (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 actually shown on top of all other instances in the same region.

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 following chapters of this manual.

Region States and UI-related Properties and Events

As mentioned in An Absolute Must-Know, the `<Manager>` creates instances of the `<Form>`. To prevent an instance from being created you cancel one of the events listed below:

<table>
<thead>
<tr>
<th>Application</th>
<th><code>&lt;Manager&gt;</code> type</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Excel</td>
<td><code>ADXExcelTaskPanesManager</code></td>
<td><code>ADXBeforeTaskPaneInstanceCreate</code></td>
</tr>
<tr>
<td>Outlook</td>
<td><code>ADXOlFormsManager</code></td>
<td><code>ADXBeforeFormInstanceCreate</code></td>
</tr>
<tr>
<td>PowerPoint</td>
<td><code>ADXPPowerPointTaskPanesManager</code></td>
<td><code>ADXBeforeTaskPaneInstanceCreate</code></td>
</tr>
<tr>
<td>Word</td>
<td><code>ADXWordTaskPanesManager</code></td>
<td><code>ADXBeforeTaskPaneInstanceCreate</code></td>
</tr>
</tbody>
</table>
An instance of the `<Form>` (further on the instance is referenced as form) is considered visible if it is embedded into the specified sub-pane of an Outlook, Excel, Word or PowerPoint window. The form may be actually invisible either due to the region state (see below) or because other forms in the same sub-pane hide it; anyway, in this case, `<Form>.Visible` returns `true`.

To prevent embedding the form into a sub-pane, you can set `<Form>.Visible` to `false` in these events:

**Table 2. Events that occur before a form instance is embedded into a sub-pane.**

<table>
<thead>
<tr>
<th>Application</th>
<th><code>&lt;Form&gt;</code> type</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>Excel</td>
<td>ADXExcelTaskPane</td>
<td>ADXBeforeTaskPaneShow</td>
</tr>
<tr>
<td>Outlook</td>
<td>ADXOlForm</td>
<td>ADXBeforeFormShow</td>
</tr>
<tr>
<td>PowerPoint</td>
<td>ADXPowerPointTaskPane</td>
<td>ADXBeforeTaskPaneShow</td>
</tr>
<tr>
<td>Word</td>
<td>ADXWordTaskPane</td>
<td>ADXBeforeTaskPaneShow</td>
</tr>
</tbody>
</table>

When the form is shown in a sub-pane, 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 the sub-pane, `<Form>.Visible` becomes `false` and the form generates the `ADXAfterFormHide` event in Outlook and the `ADXAfterTaskPaneHide` event in Excel, Word, and PowerPoint.

In accordance to the value that you specify for the `<Item>.DefaultRegionState` property, 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).

Note however that `DefaultRegionState` will work only when you show the form in a particular sub-pane for the very first time and no other forms have been shown in that sub-pane before. You can reproduce this situation on your PC by choosing `Reset Regions` in the context menu of the manager component.

You can change the state of your form at run time using the `<Form>.RegionState` property. When showing your Outlook form in certain sub-panes, you may need to show the native Outlook view or form that your form overlays; use the `ADXOlForm.ActivateStandardPane()` method.
When the region is in the hidden state, i.e. when it is collapsed to a 5px wide strip, 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>; this option is not available for the end user if you set ADXOlFormsCollectionItem.IsHiddenStateAllowed = False
- 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, see The Header and the Close Button; if the event isn’t cancelled, the Deactivate event occurs, then the pane is being removed 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 (see the figure at the right); 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 Header and the Close Button

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 <Item>.AlwaysShowHeader is set to true. The Close button is shown if <Item>.CloseButton is true. Clicking on the Close button in the form header fires the ADXCloseButtonClick event of the <Form>, the event is cancellable. You can create a Ribbon or command bar button that allows the user to show the previously closed form.
In the code below, you see how to prevent the form from being closed.

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

Repositioning the Form in the Host Application’s UI

You can let the user drag the form to a new position. This requires that all of the following conditions are met A) the form has a header, see The Header and the Close Button, B) you set ADXOlFormsCollectionItem.IsDragDropAllowed=True and C) you specify the positions in which your form can be dropped (see the ADXOlFormsCollectionItem.ExplorerAllowedDropRegions property).
Accessing a Form Instance

Add-in Express forms (panes) are based on the windowing of the corresponding Office application – Excel, Word, Outlook, and PowerPoint. At run time, Add-in Express intercepts the messages the application sends to its windows and reacts to the messages so that your form is shown, hidden, resized, etc. along with the application's windows.

In Excel 2000-2010 and PowerPoint 2000-2007, a single instance of the `<Form>` is always created for a given `<Item>` because these applications show documents in a single main window. Word is an application that shows multiple windows, and in this situation, the Word Task Panes Manager creates one instance of the task pane for every document window 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` (find more details in Form Region Instancing).

It is essential that Add-in Express panes are built on the windowing of the host application, not on the events of the application's object model. This means that getting an instance of an Add-in Express pane in a certain event may result in getting `null` (Nothing in VB.NET) if the call is issued before the pane is shown or after it is hidden. For instance, it is often the case with `WindowActivate/WindowDeactivate` in Excel, Word, and PowerPoint. Below is a list of events where Add-in Express panes may be inaccessible:

<table>
<thead>
<tr>
<th>Excel</th>
<th>WindowActivate, WindowDeactivate, WorkbookActivate, WorkbookDeactivate, NewWorkbook, WorkbookOpen, WorkbookBeforeClose</th>
</tr>
</thead>
<tbody>
<tr>
<td>PowerPoint</td>
<td>WindowActivate, WindowDeactivate, NewPresentation, AfterNewPresentation, PresentationOpen, AfterPresentationOpen, PresentationBeforeClose, PresentationClose</td>
</tr>
</tbody>
</table>

So, you may encounter a problem if your add-in retrieves a pane instance in an event above. To bypass this problem, we suggest modifying the code of the add-in so that it gets notified about a pane instance being shown or hidden (instead of getting the pane instance by handling the events above).
Use the `ADXBeforeTaskPaneShow` event of the task pane class (Excel, Word, and PowerPoint) and the `ADXOlForm.ADXBeforeFormShow` (Outlook) event to be notified about the specified pane instance being shown. When the form becomes hidden, you'll get `ADXOlForm.ADXAfterFormHide` (Outlook) and the `ADXAfterTaskPaneHide` event of the task pane class (Excel, Word, and PowerPoint).

### Controlling Form Visibility

To prevent a form from being displayed in the host application's window, you can set `<Form>.Visible to false` in the events listed in Table 2. Events that occur before a form instance is embedded into a sub-pane.

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 `<BeforeInstanceCreate>` event or set `<Form>.Visible = False in the `<BeforeFormShow>` event or the user closed it) by checking the `Visible` property of the form:

```vb
Dim Pane As A DXWordTaskPane1 = _
    TryCast(Me.AdxWordTaskPanescollectionItem1.CurrentTaskPaneInstance, _
        A DXWordTaskPane1)
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 `ADXOlFormsCollectionItem.ApplyTo` method accepting the parameter which is either `Outlook.Explorer` or `Outlook.Inspector` object;
- 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 Form**

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.

**Coloring up the Form**

By default, the background color of the form is set automatically to match the current Office 2007-2013 color scheme. To use the background color of your own in these Office versions (as well as in Office 2003), you need to set <Item>.UseOfficeThemeForBackground = True.

**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 ADXInitialize 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 ADX01FormsCollectionItem 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>.
What Window the Pane isShown for

To get an object corresponding to the host application's window that the form is shown for, use the following members:

*The properties listed below return a COM object that you must release after use.*

**Table 4. Accessing the host application's window object from Add-in Express forms**

<table>
<thead>
<tr>
<th>Application</th>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Excel</td>
<td>AXExcelTaskPane.WindowObj</td>
<td>returns Excel.Window</td>
</tr>
<tr>
<td>Outlook</td>
<td>AXOlForm.InspectorObj</td>
<td>returns Outlook.Inspector, AXOlForm.ExplorerObj returns Outlook.Explorer; these properties may also return null (Nothing in VB.NET)</td>
</tr>
<tr>
<td>PowerPoint</td>
<td>AXPowerPointTaskPane.WindowObj</td>
<td>returns PowerPoint.DocumentWindow</td>
</tr>
<tr>
<td>Word</td>
<td>AXWordTaskPane.WindowObj</td>
<td>returns Word.Window</td>
</tr>
</tbody>
</table>

**Advanced 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**

*AXExcelTaskPane* provides useful events unavailable in the Excel object model: `ADXBeforeCellEdit` and `ADXAfterCellEdit`.

**Keyboard and Focus**

*AXExcelTaskPane* 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 combination to Excel or to the pane. Sending the key combination to the pane 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 Excel Task Panes: HOWTO Samples**

The sample projects below are available for downloading from the Add-in Express web site.

**How to show and hide Advanced Excel Task Panes programmatically**

This sample project demonstrates how to set the visibility of an Advanced Excel Task Pane to make it show up.

• [Download sample project in VB.NET](#)
Advanced Outlook Regions and Advanced Office Task Panes

- Download sample project in C#

How to switch between several Advanced Excel Task Panes programmatically

This add-in project shows how you use several Excel task panes in one layout position and switch between them programmatically.

- Download sample project in VB.NET
- Download sample project in C#

How to show an Advanced Excel Task Pane dynamically

The sample project shows how to build a context-dependent Advanced Excel Task Pane. The add-in shows a task pane when cell A1 contains any value.

- Download sample project in VB.NET
- Download sample project in C#

How to resize an Excel task pane

This example shows how to resize an Advanced Task Pane. To change the form size programmatically, you set the Splitter property of the corresponding ADXX1TaskPanesCollectionItem to None. If omitting this, only the user can resize the Excel task pane using the splitter.

- Download sample project in VB.NET
- Download sample project in C#

Advanced Outlook Regions

Please see The UI Mechanics 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.ADX01FormsManager, the Outlook Forms Manager
- <Item>- AddinExpress.OL.ADX01FormsCollectionItem
- <Form>- AddinExpress.OL.ADX01Form

Context-Sensitivity of Your Outlook Form

Whenever the Outlook Forms Manager detects a context change in Outlook, it searches the ADX01FormsCollection 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 **FolderPath** and **FolderPath** properties; these properties also work for Inspector windows – in this case, the parent folder of the Outlook item is checked. An example of the folder path is "\Personal Folders\Inbox". A special value in **FolderPath** is an asterisk (**"*"**), which means "all folders". 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 **ADXOlFormsManager**.**ADXBeforeFormInstanceCreate** and **ADXOlForm**.**ADXBeforeFormShow** events in order to prevent your form from being shown (see **Accessing a Form Instance**). In addition, you can use events provided by **ADXOlForm** in order to check the current context. Say, you can use the **ADXFolderSwitch** 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 **Introducing Advanced Outlook Form and View 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 and above) 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 are not 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:

```csharp
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)
                    {
                    }
                }
            }
        }
    }
    catch
    {
    }
}
```
Form Region Instancing

The user may open multiple Explorer and Inspector windows. That is, the Outlook Forms Manager will create multiple instances of your form region class now and then. How to retrieve the form instance shown in a particular Outlook window? How to get all form instances?

**ADXOlFormsCollectionItem.GetForm()**

This method returns an instance of your form region in the specified Outlook window.

**ADXOlFormsCollectionItem.GetCurrentForm()**

This method returns an instance of your form region in the active Outlook window.

Consider the following scenarios:

- Calling `GetCurrentForm()` in the `Click` event of a Ribbon button is safe because the event can occur in the active Outlook window only; accordingly, `GetCurrentForm()` returns the form instance embedded into the Inspector (Explorer) window in which the button is clicked.
- `GetCurrentForm()` will never find e.g. an Inspector form region if an Explorer window is active;
- Some add-in or antivirus may cause the `ExplorerSelectionChange` event to fire in an inactive Explorer window; that is, using `GetCurrentForm()` in an Explorer-related event may produce a wrong result. To avoid this, use `GetForm()` or make sure that `GetCurrentForm()` is called in the active window.
From a Form Instance to the Outlook Object Model

The Outlook Forms Manager creates an instance of your form when the Outlook context matches the settings of the corresponding ADOlFormsCollectionItem.

After creating the form instance, the manager sets a number of properties providing entry points to the Outlook object model; note that these properties are not set when the form region's constructor is running. The properties are listed below. Note that the state of the COM objects returned by these properties is essential for Add-in Express functioning – you must not release them in your code because passing any of them to Marshal.ReleaseComObject() may cause Outlook to crash.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADOlForm.ExplorerObj</td>
<td>If the form is embedded (ADOlForm.Visible=True) into an Outlook Explorer window, returns a reference to the corresponding Outlook.Explorer object (a COM object). Otherwise, returns null (Nothing in VB.NET).</td>
</tr>
<tr>
<td>ADOlForm.InspectorObj</td>
<td>If the form is embedded (ADOlForm.Visible=True) into an Outlook Inspector window, returns a reference to the corresponding Outlook.Inspector object (a COM object). Otherwise, returns null (Nothing in VB.NET).</td>
</tr>
<tr>
<td>ADOlForm.FolderObj</td>
<td>If the form is embedded into an Outlook Explorer window (ADOlForm.ExplorerObj is not null), returns a reference to the Outlook.MAPIFolder object (a COM object) representing the current folder in the Explorer window.</td>
</tr>
<tr>
<td></td>
<td>If the form is embedded into an Outlook Inspector window (ADOlForm.InspectorObj is not null), returns a reference to the Outlook.MAPIFolder object (a COM object) representing the parent folder of the Outlook item which is shown in the Inspector window.</td>
</tr>
<tr>
<td>ADOlForm.FolderItemsObj</td>
<td>If the form is embedded into an Outlook Explorer window (ADOlForm.ExplorerObj is not null), returns a reference to the Outlook.Items object (a COM object) representing the collection of items of the current folder in the Explorer window.</td>
</tr>
<tr>
<td></td>
<td>If the form is embedded into an Outlook inspector window (ADOlForm.InspectorObj is not null), returns a reference to the Outlook.Items object (a COM object) representing the collection of items in the parent folder of the Outlook item which is shown in the Inspector window.</td>
</tr>
</tbody>
</table>
**Advanced Outlook Regions: HOWTO Samples**

The sample projects below are available for downloading from the Add-in Express web site.

**How to move a custom .NET form embedded into an Outlook window from one form region to another**

This sample project shows how you can change layouts of custom .NET forms embedded into Outlook windows. The add-in creates a command bar (a Ribbon tab in Office 2007-2013) and allows choosing the layout from a combo box. The form used in this sample processes the `SelectionChange` event of Outlook Explorer. Please note that you may need to change the references for the project to compile.

- [Download sample project in VB.NET](#)
- [Download sample project in C#](#)

**How to cache forms embedded into Outlook Explorer windows**

This sample project demonstrates the Advanced Outlook Form Region caching functionality available for forms embedded into Outlook Explorer windows. The developer can use this functionality to preserve form data when the user switches between folders. There are three caching options for such forms: `Non-cached`, `NewItemForEachFolder`, and `OneInstanceForAllFolders`. Forms embedded into Outlook Inspector windows are always non-cached.

- [Download sample project in VB.NET](#)
- [Download sample project in C#](#)

**How to change the size of a form embedded into an Outlook window**

This sample project demonstrates the form-sizing features available for Advanced Form Regions in Outlook 2000 - 2013. There are two options depending on the visibility of the splitter (as set by the developer): if the splitter is visible, the user can change the form size and the developer cannot. To change the Outlook form size programmatically, the developer sets the `Splitter` property to `None`.

- [Download sample project in VB.NET](#)
- [Download sample project in C#](#)

**How do you identify the instance of the form embedded into an Outlook window?**

This Outlook sample add-in shows several forms marked with a GUID. When you click a command bar button (a Ribbon button in an Outlook 2007-2013 inspector), the add-in identifies the currently active form instance and shows its GUID.
• Download sample project in VB.NET
• Download sample project in C#

Controlling the visibility of a custom form in Outlook
This sample project demonstrates the typical use of forms embedded into Outlook windows: hiding and showing the form in Explorer and Inspector windows.

• Download sample project in VB.NET
• Download sample project in C#

Several forms in the same Advanced Outlook form region
This sample project written in VB.NET demonstrates how to use several forms in the same Advanced Form Region. Click the radio button on the form to activate the corresponding ADXOlForm.

• Download sample project in VB.NET
• Download sample project in C#

How to switch between the standard Outlook Explorer view and a custom form
This sample Outlook add-in project shows how you can switch between the standard explorer view and a custom ADXOlForm form using Advanced Outlook Regions.

• Download sample project in VB.NET
• Download sample project in C#

How to use events of Reading Pane, Navigation Pane, and To-Do bar
This sample project demonstrates how you can make use of the events that are missing in Outlook:

• Show / hide the Navigation Pane (Folder List, Outlook Bar)
• Show / hide / move the Reading Pane (Preview Pane)
• Show / hide / minimize the To-Do Bar

Download sample project in C#
Events

Application-level Events

In the Add-in Express Toolbox, there is the Add Events command that creates specified events components on the module. The events components allow accessing application-level events for all Office applications. You use event handlers of such an event component to respond to events of the host application. 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 Step #13 – Handling Events of Outlook Items Object and Step #10 – Handling Excel Worksheet Events. You can use this class to implement a set of event handling rules for a given event source type. To implement another set of event handling rules for the same event source type, you add another events class to your project. For example, if the business logic is different for the Inbox and Sent Items folders in Outlook, use two events classes to handle the events of the Items collection: one for the Inbox folder and another one for the Sent Items.

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 2007-2013).

```csharp
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 ItemAdd(ByVal Item As Object)
    'TODO: Add some code
End Sub

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

Public Overrides Sub ItemRemove()
    '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.
Outlook UI Components

Outlook Bar Shortcut Manager

Outlook provides us with the Outlook Bar (Navigation Pane in Outlook 2003-2013). 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 Visual Studio, open the Add New Item dialog and choose the Outlook Options Page item to add a class to your project. This class is a descendant of System.Windows.Forms.UserControl. 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 logic 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.

Other Components

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.
Custom Toolbar Controls

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.

This feature cannot be used to customize a CommandBar control shown in the Ribbon UI.

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:

```vbnet
Private Sub AddinModule_AddinInitialize( _
    ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.AddinInitialize
    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:

```vbnet
```
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 below shows you the order of event processing by the example of the Outlook Inspector window opened by the user.

<table>
<thead>
<tr>
<th>Event Processing Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Outlook fires the built-in NewInspector event. Add-in Express traps it and fires the NewInspector event of ADXOutlookEvents.</td>
<td>ActiveInstance returns NULL.</td>
</tr>
<tr>
<td>2. ADXOutlookEvents runs your NewInspector event handlers.</td>
<td>ActiveInstance returns NULL.</td>
</tr>
<tr>
<td>3. The Toolbar Controls creates an instance of your control.</td>
<td>ActiveInstance returns NULL.</td>
</tr>
<tr>
<td>4. Outlook fires the built-in InspectorActivate event. Add-in Express handles it and fires the InspectorActivate event of ADXOutlookEvents.</td>
<td>ActiveInstance returns NULL.</td>
</tr>
<tr>
<td>5. The Toolbar Controls creates an instance of your control for the opened Inspector. ADXOutlookEvents runs your InspectorActivate event handlers.</td>
<td>ActiveInstance returns the instance of your control that was cloned from your original control.</td>
</tr>
</tbody>
</table>

Application-specific Control Adapters

All Office applications have different window architectures. 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 email 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.

Your First .NET Control on an Office Toolbar

This sample demonstrates features described in Custom Toolbar Controls.

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 Toolbar of the same sample). Now set adxMsoBarBottom to the Position property of the added command bar.

Step #1 - Adding a Control Adapter

Custom Toolbar Controls 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 \textit{(BossCheckBox)} directly to your add-in module and customize the check box in any way you like.

Step #3 - Handling Your Control

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

```vbnet
Private Sub BossCheckbox_CheckedChanged(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles AdxCommandBarAdvancedControl1.CommandBarButton_Click
    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
```

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:

```vbnet
Private Sub InitializeComponent()
    Me.components = New System.ComponentModel.Container
```
Me.AddinAdditionalModuleItem1 = New _
   AddinExpress.MSO.ADXAddinAdditionalModuleItem(Me.components)
Me.AddxOlInspectorCommandBar1 = New _
   AddinExpress.MSO.ADXxOlInspectorCommandBar(Me.components)
Me.AdxOutlookControlAdapter1 = New _
   AddinExpress.ToolbarControls.ADXOutlookControlAdapter(Me.components)
Me.BossCheckBox = New System.Windows.Forms.CheckBox
Me.AddxCommandBarAdvancedControl1 = New _
   AddinExpress.MSO.ADXCommandBarAdvancedControl(Me.components)
'
   'AdxOlInspectorCommandBar1
   '
Me.AdxOlInspectorCommandBar1.CommandBarName = _
   "AdxOlInspectorCommandBar1"
Me.AdxOlInspectorCommandBar1.CommandBarTag = _
   "77fc20e0-bf9e-47d0-997f-eb11e67f5/0a4"
Me.AdxOlInspectorCommandBar1.Controls.Add _
   (Me.AddxCommandBarAdvancedControl1)
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(CType(255, Byte), Integer), _
   CType(CType(CType(128, Byte), Integer), _
   CType(CType(CType(0, Byte), Integer), Integer)))
Me.BossCheckBox.AutoSize = True
Me.BossCheckBox.Location = New System.Drawing.Point(0, 0)
Me.BossCheckBox.Name = "BossCheckBox"
Me.BossCheckBox.TabIndex = 0
Me.BossCheckBox.Text = "BCC to my Boss"
Me.BossCheckBox.UseDefaultStyleBackColor = True
'
   'AdxCommandBarAdvancedControl1
   '
Me.AddxCommandBarAdvancedControl1.Control = Me.BossCheckBox
Me.AddxCommandBarAdvancedControl1.ControlTag = _
   "ed651259-34f1-4d00-8716-e56ccf0118d4"
Me.AddxCommandBarAdvancedControl1.Temporary = True
Me.AddxCommandBarAdvancedControl1.UpdateCounter = 3
'
   'AddinModule
   '
Me.AddinName = "MyAddin"
Me.SupporttedApps = AddinExpress.MSO.ADXOfficeHostApp.ohaOutlook
Step #5 - Register and Run Your Add-in

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

Hello John,

I have just found a nice solution. Please have a look at [http://www.add-in-express.com](http://www.add-in-express.com) and contact our boss to discuss it.

Unfortunately, they don’t publish trial versions :-(

Thank you,
Bob
Deploying Office Extensions

How to install the Office extension you developed to another machine? In this section, we describe your ways to all deployment technologies supported by Add-in Express.

- All Deployment Technologies at a Glance
- Deployment: Things to Consider
- Creating MSI Installers
- ClickOnce Deployment
- ClickTwice :) Deployment
- Deployment Step-by-steps
## All Deployment Technologies at a Glance

### Table 5. Deployment technologies. Links to step-by-step instructions.

<table>
<thead>
<tr>
<th>How you install the Office extension</th>
<th>A per-user COM add-in, RTD server, Smart tag, or Excel UDF</th>
<th>A per-machine COM add-in or RTD server</th>
</tr>
</thead>
<tbody>
<tr>
<td>A standard user (per-user installation) or administrator (per-machine installation) runs the installer from a CD/DVD, hard disk or local network location</td>
<td>Windows Installer ClickOnce ClickTwice :)</td>
<td>Windows Installer ClickTwice :)</td>
</tr>
<tr>
<td>A corporate admin uses Group Policy to install your Office extension for a specific group of users in the corporate network; the installation and registration occurs when a user logs on to the domain.</td>
<td>Windows Installer</td>
<td>N/A</td>
</tr>
<tr>
<td>A standard user (per-user installation) or administrator (per-machine installation) runs the installer by navigating to a web location or by clicking a link.</td>
<td>ClickOnce ClickTwice :)</td>
<td>ClickTwice :)</td>
</tr>
</tbody>
</table>

### Table 6. Deployment technologies. Short descriptions.

<table>
<thead>
<tr>
<th>Windows Installer</th>
<th>You create a regular .MSI installer to install per-user and per-machine Office extensions. To update your Office extension, you uninstall its current version and install the new one. A per-user add-in is installed by an end user; a per-machine add-in requires administrative permissions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClickOnce</td>
<td>This technology is targeted at non-admin installations; only per-user Office extensions can be installed in this way. When the user updates your Office extension, its previous version is uninstalled automatically.</td>
</tr>
<tr>
<td>ClickTwice :)</td>
<td>This is a custom MSI-based Web deployment technology. ClickTwice :) allows standard users and admins to run MSIs from the web (Internet and Intranet) for installing and updating per-user and per-machine Office extensions.</td>
</tr>
</tbody>
</table>

### Table 7. Deployment technologies. Detailed Comparison.

<table>
<thead>
<tr>
<th>Feature</th>
<th>ClickOnce</th>
<th>Windows Installer</th>
<th>ClickTwice :)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update from the Web</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Post-installation rollback</td>
<td>Via Add/Remove Programs</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Security</td>
<td>Grants only permissions necessary for the</td>
<td>Grants Full Trust</td>
<td>Grants Full Trust by</td>
</tr>
<tr>
<td>permissions granted</td>
<td>application (deploying COM add-ins, it always requires Full Trust).</td>
<td>by default</td>
<td>default</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>---------------------------------------------------------------------</td>
<td>------------</td>
<td>---------</td>
</tr>
<tr>
<td>Security permissions required</td>
<td>Internet or Intranet Zone (Full Trust for CD-ROM installation)</td>
<td>Standard user or Administrator</td>
<td>Standard user or Administrator</td>
</tr>
<tr>
<td>Installation-time user interface</td>
<td>Single prompt</td>
<td>Multipart Wizard</td>
<td>Multipart Wizard</td>
</tr>
<tr>
<td>Installation of shared files</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Installation of drivers</td>
<td>No</td>
<td>Yes (with custom actions)</td>
<td>Yes (with custom actions)</td>
</tr>
<tr>
<td>Installation to Global Assembly Cache</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Installation for multiple users</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Add entry to Start menu</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Add entry to Startup group</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Add entry to Favorites menu</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Register file types</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Install time registry access</td>
<td>HKEY_LOCAL_MACHINE (HKLM) accessible only with Full Trust permissions</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Binary file patching</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Installation location</td>
<td>ClickOnce Application Cache</td>
<td>Changed by the user during the installation</td>
<td>Changed by the user during the installation</td>
</tr>
</tbody>
</table>
Deployment: Things to Consider

How Your Office Extension Is Registered

A setup project created as described in Creating MSI Installers uses adxregistrator.exe as a custom action. When you run the installer and adxregistrator.exe is invoked, it performs the following steps:

- loads .NET Framework,
- creates an instance of the add-in module,
- invokes the registration code provided by the Add-in Express module; there is a special module for each Office extension type, see Add-in Express Modules.

When doing all the things above, adxregistrator.exe writes them into a log file, its default location is {user profile}\Documents\Add-in Express\adxregistrator.log.

What follows below is a description of the process and how you can customize it. See also Troubleshooting add-in registration.

Specifying the Assembly to register

adxregistrator.exe supports /install and /uninstall switches. They accept a string parameter containing the file name of the assembly that is to be registered/unregistered.

adxregistrator.exe /install="MyAddin1.dll"

All add-in files including dependencies and the Add-in Express assemblies must be located in the folder where adxregistrator.exe is run.

Registration-time Privileges

A COM add-in has two sides: it is a COM class and an Office add-in at the same time. Both sides of the COM add-in require proper registration in the Windows Registry.

The add-in side of a COM add-in relates only to Office: a per-user COM add-in is denoted by False in the RegisterForAllUsers property of its add-in module and it is registered in HKCU. A per-machine add-in has True in the RegisterForAllUsers property of its add-in module and it is filed down in HKLM. The exact registry paths are given in Registry Keys.
Before you modify the `RegisterForAllUsers` property, you must unregister the add-in project on your development PC and make sure that `adxloader.dll.manifest` is writable.

But the **COM class** side of a COM add-in, the COM object implemented by the add-in module and corresponding to the add-in as a whole, must be registered, too. It can be registered either in HKCU or in HKLM (the same as the add-in side). This is controlled by the `/privileges` switch supported by `adxregistrator.exe`. The switch accepts two values: `admin` and `user`. Misspelling the value or omitting the switch results in registering the COM object of the COM add-in in HKLM and this requires administrative permissions.

The need to register both, the COM class and the Office add-in itself, creates four possible combinations of settings you can specify in `RegisterForAllUsers` and in the `/privileges` switch of `adxregistrator.exe`. The two combinations below are recommended:

- **per-user add-ins:**

  add-in module:  
  `RegisterForAllUsers = False`

  `adxregistrator.exe`: `/privileges=user`

- **per-machine add-ins:**

  add-in module:  
  `RegisterForAllUsers = True`

  `adxregistrator.exe`: `/privileges=admin`

Please note that for a per-machine add-in, all users of the add-in must have appropriate permissions for the folder the add-in is installed to. The Add-in Express team recommends installing such an add-in to **Program Files**.

**CLR version to use while registering**

By default, `adxregistrator.exe` loads the latest version of the .NET Framework installed on the PC. This can be a problem if your assembly uses version-sensitive components. To bypass this, you can use the `/CLRVersion` switch that accepts a string value in the format below:

`major[.minor].build`

The value you assign to a switch is processed as described below:
• `/CLRVersion="2.0.50727"` refers to the specified build of the .NET Framework. If the build with the exact build number is not installed on the PC, the newest of all .NET Framework versions installed on the PC will be loaded. This will also occur if any other build of .NET Framework 2.0 is installed on the PC.

• `/CLRVersion="2.0"` refers to any build of .NET Framework 2.0. In a hypothetical scenario with the now non-existing .NET Framework 2.1 installed, using `/CLRVersion="2.0"` will result in loading the latest version of the .NET Framework installed on the PC (after the registrator does not find .NET Framework 2.0).

• `/CLRVersion="2"` refers to any build of .NET Framework 2.

If the specified version of the .NET Framework is not installed on the PC, the newest of all .NET Framework versions installed on the PC is loaded instead.

Creating Add-in Module instance while registering the add-in

After the CLR is loaded, the registrator creates an `AppDomain`, loads the assembly specified by you (see Specifying the Assembly to register), creates an instance of the Add-in Express module defined in the assembly and run the registration code provided by every module of Add-in Express. If the assembly includes several Add-in Express modules (=several Office extensions), the registrator processes all of them in turn.

Registering. An Important Note

Creating an instance of the module invokes the module’s constructor. It means that you should foresee the situation in which the module is created outside of the Office application. If you don’t, a run-time exception may prevent your Office extension from being registered or unregistered. The simplest way to bypass this is not to write custom code in the constructor of the module. Instead, you can use the events the module provides.

Do not write custom code in the constructor of the module.

Note that if a variable in the module is declared on the class level, the initializer of the variable is called even before the constructor of the module. That is why all the reasoning for not using custom code in the module’s constructor does apply to initializers.

Do not use initializers of complex-type variables declared on the class level in the module.
Running the Registration Code

Every Add-in Express module provides a static (Shared in VB.NET) method with the `ComRegisterFunction` attribute applied. That method invokes the registration code defined in the base class of the module. Note that if you create a custom static (Shared in VB.NET) method and apply `ComRegisterFunction` to it, the method will be executed during registration. The `ComUnregisterFunction` attribute is processed in a similar fashion; if this attribute is applied to a method, the method will be called while unregistering the extension. There's no way to predict or change the order in which methods having such attributes are called.

Get details about add-in registration/unregistration

The process of registration/unregistration is documented in a log file, the default location of which is `{user profile}\Documents\Add-in Express\adxregistrator.log`. The registrar supports the `/LogFileLocation` switch that allows you to specify the path and file name of the log file. The log file will not be generated if you use `/GenerateLogFile=false`; omitting that switch means the file will be generated.

When specifying the path to the log file, you can refer to a system folder using a widespread notation, a sample of which is `%UserProfileFolder%`. Below is the list of supported folder IDs (please find their definitions [here](#)):

- `ProgramFilesX64Folder`
- `RoamingAppDataFolder`
- `DesktopFolder`
- `PersonalFolder`
- `InternetCacheFolder`
- `LocalAppDataFolder`
- `AppDataFolder`
- `DocumentsFolder`
- `MyDocumentsFolder`
- `UserProfileFolder`
- `ProgramFilesFolder`
- `CommonProgramDataFolder`
- `PublicDocumentsFolder`
- `PublicDesktopFolder`
- `ProgramFilesX64CommonFolder`
- `ProgramFilesCommonFolder`
- `Temp`
- `TempFolder`

The supported macro characters are as follows: `<`, `&`, `[]`, `$$`, `%`. 
Exit code when Registering

If a custom action returns a non-zero exit code, the .MSI installer produces nasty dialogs that may scare the end user and produce extra problems for the developer. That is why the default value of the \ReturnExitCode switch supported by the registrator is false. Nevertheless, in custom scenarios you may want to be notified about problems as soon as possible. Set the switch to true and get a value that you can decipher using the information supplied here.

How Your Office Extension Loads Into an Office Application

When a user starts an Office application, the application locates the Add-in Express loader, which is an unmanaged DLL that loads the add-in assembly, see Add-in Loading Process. The following figure shows the basic architecture of Add-in Express based add-ins.

Add-in Loading Process

The following steps occur when a user starts an Office application.
1. The application locates the registry entries that identify your add-in. These records are created as part of the registration process. See also Locating COM Add-ins in the Registry and How Your Office Extension Is Registered.

2. If the application finds the corresponding registry entries, it loads either adxloader.dll (32bit Office) or adxloader64.dll (64bit Office). The installer of your add-in deploys these DLLs to the add-in's installation folder. See also Add-in Express Loader.

3. The loader reads the manifest, which is located in the same folder and locates the add-in assembly. If the manifest specifies this, a log file is created at the specified location. See also Add-in Express Loader Manifest, Get details about add-in loading and Troubleshooting add-in loading.

4. The loader initializes CLR, creates an AppDomain using the configuration file specified by the manifest, loads your assembly into the domain, and creates an instance of your add-in module. See also Add-in Express Loader Manifest.

5. When one of the startup events is received on the IDTExtensibility2 and IRibbonExtensibility COM interfaces, the loader translates the call to the AddinInitialize and OnRibbonBforeCreate events of the add-in module, respectively, and in this way your assembly starts functioning as an Office add-in.

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

We use these terms to name the registry keys described below:

- add-in key
- ProgId key
- CLSID key

Depending on the value of the RegisterForAllUsers property of the add-in module (to access it in the Properties window, click the designer surface of the module), the main registry entry of a COM add-in, the add-in key is:
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`.

Before you modify the `RegisterForAllUsers` property, you must unregister the add-in project on your development PC and make sure that `adxloader.dll.manifest` is writable.

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

The other keys – the `ProgId` key and `CLSID` key – are demonstrated in the figure below:
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 + ""

If an Excel add-in is turned off in the Excel Add-ins dialog, see the path to the add-in’s loader in:

HKEY_CURRENT_USER\Software\Microsoft\Office\{version}.0\Excel\Add-in Manager

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 based Office extension. That software is called a shim. A shim is an unmanaged DLL that isolates 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 pre-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 to be registered as Office extensions. The loader’s files (adxloader.dll, adxloader64.dll and adxloader.dll.manifest) must 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. The source code of the loader is available on request for Premium customers only.

You can find more background info in Insight of Add-in Express Loader.

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.
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 attribute; relative paths and folder constants are acceptable, see Get details about add-in registration/unregistration. 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. 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 (to access it in the Properties window click the designer surface of the add-in module or RTD server module). 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.

**Before you modify the RegisterForAllUsers property, you must unregister the add-in project on your development PC and make sure that adxloader.dll.manifest is writable.**

Use the minOfficeVersionsupported attribute to let the add-in load in supported Office versions only. Not specifying this attribute means that your add-in will be loaded in any Office version from 2000 to the latest. Loading the add-in using other conditions is discussed in How to load your Office COM add-in on condition.

For more information about setting clrVersion, see CLR version to use while registering.

The use of the configFileName attribute is described in Configuring an Add-in.

Note that 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.
Get details about add-in loading

If the manifest requires creating a log file (see the `generateLogFile` attribute at Add-in Express Loader Manifest), the log file (`adxloader.log`) 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 when the Office application loads your add-in.

Per-user or Per-machine?

An Office extension can be per-user or per-machine. By default, you Add-in Express project creates a per-user Office extension.

Neither the `Everyone` option of an MSI installer nor the `InstallAllUsers` property of your setup project relate to installing Office extensions in the "for all users on the PC" way. Please see InstallAllUsers Property of the Setup Project for details.

To let your Office extension work for all users on the machine, you need to set the `RegisterForAllUsers` property of the corresponding Add-in Express module (to access it in the Properties window, click the designer surface of the module).

Before you modify the `RegisterForAllUsers` property, you must unregister the add-in project on your development PC and make sure that `adxloader.dll.manifest` is writable.

Note that if the module doesn't expose the `RegisterForAllUsers` property, then the Office extension you create cannot be registered for all users on the PC; this is by design from Microsoft. The table below describes the availability of the "for all users" registration for Office extensions.

<table>
<thead>
<tr>
<th></th>
<th>Per-user</th>
<th>Per-machine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Registers to HKCU Standard User permissions</td>
<td>Registers to HKLM Requires administrative permissions</td>
</tr>
<tr>
<td>COM Add-in</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>Excel RTD Server</td>
<td>+</td>
<td>–</td>
</tr>
<tr>
<td>Excel Automation Add-in</td>
<td>+</td>
<td>–</td>
</tr>
<tr>
<td>Excel XLL Add-in</td>
<td>+</td>
<td>–</td>
</tr>
<tr>
<td>Smart Tag</td>
<td>+</td>
<td>–</td>
</tr>
</tbody>
</table>

Per-machine extensions cannot be installed via ClickOnce, which is a deployment technology targeted to non-administrator-privileges installations.
Installing and Registering

You cannot deploy an Office extension using XCOPY because this does not create required registry entries.

When you run the installer on the target machine, the question arises: where to install the add-in? Note that per-user extensions are called so because a standard user is able to install them; that means that the user may install a per-user extension to any folder accessible for the user. Also note that ClickOnce installers always install to ClickOnce Application Cache. A per-machine extension requires administrative permissions and only administrators can install it; the target folder must be accessible by all users of the extension.

Although, in the general case, you cannot prevent the user from choosing an incorrect folder, you can provide a valid default installation location. If you create a setup project using Add-in Express (see Creating MSI Installers), the setup project wizard analyzes RegisterForAllUsers of the Add-in Express module used in your project and creates a setup project that is ready to install the files mentioned in Files to Deploy to the following default locations:

<table>
<thead>
<tr>
<th>RegisterForAllUsers = True</th>
<th>RegisterForAllUsers is missing</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ProgramFilesFolder][Manufacturer][ProductName]</td>
<td>[AppDataFolder][Manufacturer][ProductName]</td>
</tr>
</tbody>
</table>

Still, installing an Office extension isn't enough. To get loaded to the corresponding Office application, your Office extension must be described correctly in the Windows Registry; see Registry Keys for exact registry locations. Add-in Express writes all required information to the correct registry locations so that you usually don't even think about this.

Permissions Required

An Office extension with RegisterForAllUsers set to False or no such property, writes to HKCU during registration, thus it can be registered by a standard user. Since writing to HKLM requires administrative permissions, only administrators can install (and register) a COM add-in or RTD server for all users on the PC; only these Office extension types have the RegisterForAllUsers properties in their modules.

However, before being registered, the Office extension must be installed. Only the user having corresponding permissions can do this. Of course, this applies to any other software.

Files to Deploy

The tables below contain minimal sets of files required for your Office extension to run.
Office add-ins, XLL add-ins

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddinExpress.MSO.2005.dll</td>
<td>Command bar and Ribbon controls, COM add-in and XLL</td>
</tr>
<tr>
<td>Interop assemblies</td>
<td>All interops required for your add-in</td>
</tr>
<tr>
<td>adxlabel.dll</td>
<td>32-bit loader; required for Office 2000-2007, and 32bit Office 2010-2013</td>
</tr>
<tr>
<td>adxlabel64.dll</td>
<td>64-bit loader; required for 64-bit Office 2010-2013</td>
</tr>
<tr>
<td>adxlabel.dll.manifest</td>
<td>Loader manifest</td>
</tr>
<tr>
<td>adxregistrator.exe</td>
<td>Add-in registrator</td>
</tr>
<tr>
<td>intResource.dll, intResource64.dll</td>
<td>Ensures compatibility between various Add-in Express based add-ins. If not available in the add-in folder, Add-in Express unpacks it to the Temporary Files folder and loads into the host application.</td>
</tr>
</tbody>
</table>

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

Excel Automation add-ins

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddinExpress.MSO.2005.dll</td>
<td>Excel automation add-ins</td>
</tr>
<tr>
<td>Interop assemblies</td>
<td>All interops required for your add-in</td>
</tr>
<tr>
<td>adxregistrator.exe</td>
<td>Add-in registrator</td>
</tr>
</tbody>
</table>

RTD servers

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddinExpress.RTD.2005.dll</td>
<td>Excel RTD Server</td>
</tr>
<tr>
<td>adxlabel.dll</td>
<td>32-bit loader; required for Office 2002-2007, and 32bit Office 2010-2013</td>
</tr>
<tr>
<td>adxlabel64.dll</td>
<td>64-bit loader; required for 64-bit Office 2010-2013</td>
</tr>
<tr>
<td>adxlabel.dll.manifest</td>
<td>Loader manifest</td>
</tr>
<tr>
<td>adxregistrator.exe</td>
<td>Add-in registrator</td>
</tr>
</tbody>
</table>

Smart tags

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddinExpress.SmartTag.2005.dll</td>
<td>Smart Tag</td>
</tr>
<tr>
<td>adxlabel.dll</td>
<td>32-bit loader; required for Office 2002-2007, and 32bit Office 2010-2013</td>
</tr>
</tbody>
</table>
**Publishing from the Command Prompt**

Add-in Express provides a command line tool that you use to automate the process of publishing your add-in projects. That tool is `adxPublisher.exe` located in the `Bin` folder of the Add-in Express installation folder. The utility is provided with the default configuration file; the file is called `adxpublisher.exe.config`; the file provides options and their descriptions.

To use the utility, you copy the `.config` file to an appropriate location and modify it there. The `adxpublisher.exe.config` file contains two sections, which we refer to as the `ClickOnce` section and the `ClickTwice` section. The sections contain settings for the corresponding deployment technologies.

You run the utility in this way:

- `Adxpublisher.exe /OutputType=ClickOnce`
- `Adxpublisher.exe /OutputType=ClickTwice`

If you don’t specify the parameter, the utility will use the first section found.

<table>
<thead>
<tr>
<th>adxloader64.dll</th>
<th>64-bit loader; required for 64-bit Office 2010-2013</th>
</tr>
</thead>
<tbody>
<tr>
<td>adxloader.dll.manifest</td>
<td>Loader manifest</td>
</tr>
<tr>
<td>adxregistrator.exe</td>
<td>Add-in registrator</td>
</tr>
</tbody>
</table>
Creating MSI Installers

Running the Setup Project Wizard

To help you create an installer for your Office extension, Add-in Express provides the setup project wizard accessible via menu Project | Create Setup Project in VS. Another way to run the wizard is shown in the screenshot.

Let's run the setup project wizard for the sample project described in Your First Microsoft Office COM Add-in:

The setup project wizard supports creating setup projects for InstallShield 2009 Professional and higher. The Limited and Express editions of InstallShield are not supported.
Creating MSI Installers

Welcome to the Add-in Express for Microsoft Office and .NET Setup Project Wizard

- Visual Studio Installer
- InstallShield
- WiX

- Title: MyAddin1
- Description: MyAddin1
- Product name: MyAddin1
- Company: My Company

Welcome to the Add-in Express for Microsoft Office and .NET Setup Project Wizard

- Localization: Default
- File name: MyAddin1Setup(1.0.0)
- Output directory: D:\Projects\MyAddin1\MyAddin1\MyAddin1Setup\1.0.0
The wizard creates and adds the following setup project to the solution:

![Solution Explorer](image)

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

The wizard creates the following entries in the Application Folder (see menu View | Editor | File System):
Also, the following custom actions are created:
Creating a Visual Studio Setup Project 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 adds a new 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.

**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 / Custom Actions` 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="{add-in assembly name}.dll" /privileges={user OR admin}
  ```

- **Rollback**
  
  ```
  /uninstall="{add-in assembly name}.dll" /privileges={user OR admin}
  ```
Creating MSI Installers

- Uninstall

  /uninstall="{add-in 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 admin.

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 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 prerequisites because installing a prerequisite 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
  ```

The executable – adxPatch.exe – also provides the /RunActionsAsInvoker switch. If set to true (default), the switch specifies that adxRegistrar.exe used as a custom action will be run with the privileges of the user who launches the installer.

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 prerequisites, you will need to run setup.exe.

Creating a WiX Project

The below is a minimal introduction to using WiX to deploy an Office extension created with Add-in Express. As you understand, it is not possible to describe all the features provided by WiX. We recommend that you study the WiX manual at http://wix.sourceforge.net/manual-wix3/main.htm.

The setup project wizard creates a WiX project containing just one file called Product.wxs. The file is the source of the configuration information (in the XML format) for the WiX binaries mentioned in the References section of the WiX project.

The product to be installed is described in the tag Product.

```xml
<Product
  Id="4bb88615-45a4-41ef-97ba-3d03337a3b0a"
  Name="MyAddin1"
  Language="1033"
  Version="1.0.0"
  Manufacturer="My Company"
  UpgradeCode="4bb88615-45a4-41ef-97ba-3d03337a3b0b"
  Codepage="1252">
  <Package
    AdminImage="no"
    Comments="MyAddin1"
    Compressed="yes"
    Description="MyAddin1"
    InstallerVersion="200"
    InstallScope="perUser"
    Languages="1033"
    Manufacturer="My Company"
```
The set of important tags describes custom actions that register and unregister your Office extension:

```xml
<Binary Id="adxregistrator_exe"
   Sourcefile="$(var.ADX_PATH)\Redistributables\adxregistrator.exe" />

</Binary>

<Condition Message="This setup requires the .NET Framework 4.0. Please install the .NET Framework and run this setup again.">
   <![CDATA[Installed OR NETFRAMEWORK40CLIENT]]>
</Condition>

<PropertyRef Id="NETFRAMEWORK40CLIENT"/>
```

The "adxregistrator.exe" custom action.

```xml
<CustomAction Id="_87A0494E_2243_4197_A698_E5D4350CC2C8"
   BinaryKey="adxregistrator_exe"
   Execute="deferred"
   Impersonate="yes"/>
```

The "adxregistrator.exe" custom action.

```xml
<CustomAction Id="_74769CB0_2F84_42A6_A9FE_B166FADD2B74"
   BinaryKey="adxregistrator_exe"
   Execute="rollback"
   Impersonate="yes"/>
```

The "adxregistrator.exe" custom action.

```xml
</InstallExecuteSequence>
```
And finally, there're tags describing files that will be delivered to the target PC. The minimum set of files includes:

- your assembly
- loader manifest
- 32bit loader
- 64bit loader
- Interops for all Office applications supported by your add-in
- Add-in Express assemblies e.g. AddinExpress.MSO.2005.dll

```xml
<Component
  Id="comp_3F676BCF_7D32_43E9_8A42_D60F96167397"
  Guid="15eb145d-2f84-4a4e-a0be-6e81c14db478"
  Permanent="no" SharedDllRefCount="no" Transitive="no">
  <RegistryKey Root="HKCU" Key="Software\[Manufacturer]\[ProductName]">
    <RegistryValue Type="string" Name="Installed" Value="[INSTALLFOLDER]"
      KeyPath="yes" />
  </RegistryKey>
  <File
    Id="_AEBCC10D_6306_4241_BBCC_46AF9341E288" DiskId="1" Hidden="no"
    ReadOnly="no" System="no" Vital="yes" Compressed="yes"
    Name="adxloader64.dll"
    Source="$(var.MyAddin1.ProjectDir)Loader\adxloader64.dll" />
  <File
    Id="_17044247_6624_4780_B6B1_4534C7644A7F" DiskId="1" Hidden="no"
    ReadOnly="no" System="no" Vital="yes" Compressed="yes"
    Name="adxloader.dll.manifest"
    Source="$(var.MyAddin1.ProjectDir)Loader\adxloader.dll.manifest" />
  <File
    Id="_2F2D8089_C07B_4C24_8F53_50B37DFF036C" DiskId="1" Hidden="no"
    ReadOnly="no" System="no" Vital="yes" Compressed="yes"
    Name="adxloader.dll"
    Source="$(var.MyAddin1.ProjectDir)Loader\adxloader.dll" />
</Component>
```
Please refer to the WiX manual for more information, see http://wix.sourceforge.net/manual-wix3/main.htm.
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. 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.

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.

ClickOnce. 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.

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 target URL of the deployment manifest (originally, 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.

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 Installer Files**

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:

![Preferences Dialog]

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.

Deployment manifest – &lt;SolutionFolder&gt;/Publish/&lt;projectname&gt;.application
Application manifest - &lt;SolutionFolder&gt;/Publish/&lt;ProjectVersion&gt;/&lt;ProjectName&gt;.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.

ClickOnce. On the Target PC

Installing: User Perspective

The user browses the deployment manifest (&lt;projectname&gt;.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 Application 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.

After installing the add-in, 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 Application 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 or 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](https://en.wikipedia.org/wiki/ClickOnce) article from Wikipedia states that Firefox allows ClickOnce-based installations too, but this was neither tested nor even verified.
Customizing ClickOnce installations

You can add a ClickOnce module to your add-in project and handle the `ADXClickOnceModule.OnClickOnceAction` event:

```vbnet
Private Sub ClickOnceModule1_OnClickOnceAction(sender As System.Object, _
action As AddinExpress.MSO.ADXClickOnceAction) _
Handles MyBase.OnClickOnceAction
    If action.HasFlag(AddinExpress.MSO.ADXClickOnceAction.Register) Then
        'registering
        If action.HasFlag(AddinExpress.MSO.ADXClickOnceAction.Install) Then
            'registering while installing for the first time
        ElseIf action.HasFlag(AddinExpress.MSO.ADXClickOnceAction.Update) Then
            'registering an update
        End If
    ElseIf action.HasFlag(AddinExpress.MSO.ADXClickOnceAction.Unregister) Then
        'unregistering
        If action.HasFlag(AddinExpress.MSO.ADXClickOnceAction.Uninstall) Then
            'unregistering while uninstalling
        ElseIf action.HasFlag(AddinExpress.MSO.ADXClickOnceAction.Update) Then
            'unregistering before updating
        End If
    ElseIf action.HasFlag(AddinExpress.MSO.ADXClickOnceAction.PreviousVersion) Then
        'unregistering before reverting to the previous version
    End If
End Sub
```

Note that ClickOnce doesn't provide any opportunity to customize or hide dialogs and messages shown while the user updates your add-in.
ClickTwice :) Deployment

ClickTwice is designed to help you deploy and update per-user and per-machine add-ins over the web. This technology allows using your favorite deployment tool to create an .MSI installer. For instance you can develop your setup project in Visual Studio, see Creating MSI Installers.

Introduction to ClickTwice

You start with publishing the installer. Open the Publish dialog to specify the .MSI file to be published, the URL from which the add-in and updates will be installed, provide certificate info, select prerequisites and choose options. You can customize the installation process by providing a custom action DLL containing a ClickTwice module; that class provides a programmatic interface to the ClickTwice functionality.

Clicking the Publish button prepares the files required for your add-in to be deployed and organizes them in several folders within the specified folder (local). There are several important files to note:

- A bootstrapper, which is generated only if prerequisites are defined. It downloads and installs the prerequisites from the specified URL. If prerequisites are installed, the setup.exe downloads and runs the downloader (see below).
- A downloader; it downloads the .MSI file and runs it.
- An XML file containing version information about updates.

You copy the complete folder structure including the files described above and prerequisites to a location, which is accessible by the end user via the specified URL. Then you provide the user with a link pointing to the bootstrapper or downloader (depending on whether prerequisites are selected or not). The user clicks the link, the browser downloads the bootstrapper or downloader and starts it, this ends up with the downloader downloading the .MSI and running it. Finally, the add-in gets installed.

Publishing an update modifies the XML file and puts the new bootstrapper, downloader and .MSI to separate folders in the same local folder. You copy the folder to the location accessible by end users. When your add-in invokes the CheckForMsiUpdates() method, it downloads the XML file and checks the version information. If there’s a new update, it returns the URL of the bootstrapper or downloader. You download the file and start it.

Publishing with ClickTwice :)

Build the add-in project and choose Publish ADX Project in the project context menu or in the Project menu. This opens the dialog window shown in the screenshot below.
Installer File

Specifying the path to the .MSI installer in the Publish dialog fills in some fields in the upper part of the dialog.
Publishing Location

You may publish the installer to a file share or FTP server; publishing to an IIS is not supported. When publishing the installer, the *Publishing location* can be a file path or a path to an FTP server. By default, the Publish wizard suggests publishing your application to the *MSIPublish* subfolder in the project directory. In *Publishing location*, enter the location using one of the following formats:

- To publish to a file share, enter the path using either a UNC path such as \Server\ApplicationName or a file path, say C:\Deploy\ApplicationName.
To publish to an FTP server, enter the path using the format `ftp://ftp.domain.com/ApplicationName`.

**Installation URL**

The location from which users download and run the installer may differ from the location where you initially published it. For example, in some organizations, a developer might publish an application to a staging server, and then an administrator can move the application to a web server. In this case, you can use the **Installation URL** field to specify the Web server to which users will go to download the installer. This step is necessary for Add-in Express to know where to check for updates. In **Installation URL**, enter the installation location using a fully qualified URL in the format `http://www.domain.com/ApplicationName`, or a UNC path using the format `\Server\ApplicationName`.

If **Publishing location** and **Installation URL** are the same, **Installation URL** will be empty when you open the **Publish wizard** next time.

**Icon File**

You can specify an icon in the **Icon file** field. The icon will be shown in the downloader window, which is displayed when the installer is downloaded from the installation location.

**Certificate**

To sign the installation files, browse for the existing certificate file or click **New** to create a new one. Enter the password for the certificate (optional).

**Preferences**

Click the **Preferences** button to open the **Preferences** dialog.
The dialog allows you to specify if the Windows installer UI will be shown during installation / uninstallation. Also, it allows specifying a URL of a web page to be returned by the `CheckForUpdates` method, see Updating an Office Extension via ClickTwice :). This allows implementing custom update logics, welcome pages, information pages, etc.

Prerequisites

Open the *Prerequisites* dialog and select the prerequisites required by your Office extension.

You must choose the following prerequisites for installing on a clean PC:

- the .NET Framework version you used when developing your Office extension;
- Windows Installer 3.1 or 4.5.
Clicking Publish

Everything is ready; let's click the *Publish* button.

This generates files discussed below.
**Files Generated by ClickTwice :)**

In the process of publishing controlled by the *Publish* wizard, you specify the `.MSI` installer to be published as well as the location where subsequent updates of your Office extension will be located. The wizard generates the folder structure demonstrated in the screenshot below:

![Folder Structure Screenshot](image)

Below, we describe every folder shown in the screenshot.
Folder MSIPublish

The root folder of the folder structure created by the Publish wizard is the MSIPublish folder; you can specify any other folder, of course. version-info.xml describes all updates available for the given product. A sample version-info.xml is shown below:

```xml
<?xml version="1.0" encoding="utf-8"?>
<application name="myaddin1">
  <product language="1033">
    <version name="1.0.0" installationUrl="http://www.MySite.com/Updates/"
      productCode="{4A23B0EC-70BA-431D-BB27-C1D8F820F534}"
      updateType="bootstrapper">
      <files msi="MyAddin1Setup(1.0.0)">
        <file>MyAddin1Setup(1.0.0).msi</file>
      </files>
    </version>
    <preferences>
      <showInstallUI>true</showInstallUI>
      <showUninstallUI>false</showUninstallUI>
      <webPage>
        <webPage>
      </preferences>
    </version>
    <version name="1.0.1" installationUrl="http://www.MySite.com/Updates/"
      productCode="{9465C230-1A48-4D52-AC34-99EFFE8EB2C4}"
      updateType="bootstrapper">
      <files msi="MyAddin1Setup(1.0.1)">
        <file>MyAddin1Setup(1.0.1).msi</file>
      </files>
    </version>
    <preferences>
      <showInstallUI>true</showInstallUI>
      <showUninstallUI>false</showUninstallUI>
      <webPage>
        <webPage>
      </preferences>
    </version>
  </product>
</application>
```
Language-specific Folders (1033 etc.)

Folder 1033 in the screenshot below is named according to the language code of the installer UI. A list of language codes (or Locale Ids) can be found [here](#).

Version-specific Folders (1.0.0 & 1.0.1)

These folders are named according to the version of your .MSI installer; see the Version property of the setup project in VS. You see the content of such a folder in the screenshot below:

The folder includes the following files:

- The .MSI installer you specified.
- setup.exe; it is an unmanaged executable, which is generated only if prerequisites are specified. It downloads and installs the prerequisites from 
  \(\text{Installation URL}/\{\text{Language code}\}/\{\text{Version}\}/\text{prerequisites}\). When prerequisites are installed, the setup.exe runs the downloader (see below).
• An executable called downloader; it downloads the .MSI file from \{Installation URL}\{(Language code)\{(Version)\} and runs it. It is launched either by setup.exe (after installing prerequisites) or by the user (if no prerequisites are used). The file name of the downloader is set to the name of your project, such as myaddin1.exe.

Updating an Office Extension via ClickTwice :)

Every Add-in Express module provides the CheckForMSIUpdates method. When your Office extension calls it, the version_info.xml is downloaded via HTTP and parsed. If there are no updates, CheckForMSIUpdates returns an empty string. If there are new updates, CheckForMSIUpdates finds the latest update and returns either the URL of the corresponding setup.exe (if it exists) or the downloader.

To implement custom update logics, welcome pages, information pages, etc., you may choose CheckForMSIUpdates to return a URL of your choice; see Preferences.

The code sample below demonstrates installing a new version of your COM add-in programmatically; the user clicks a Ribbon button to initiate the process.

```csharp
private void adxRibbonButton1_OnClick(object sender, AddinExpress.MSO.IRibbonControl control, bool pressed)
{
    if (this.IsMSINetworkDeployed() && this.IsMSIUpdatable())
    {
        string updateUrl = this.CheckForMSIUpdates();
        if (!String.IsNullOrEmpty(updateUrl))
        {
            if (MessageBox.Show("A new version of the add-in was detected. " +
                                "Would you like to install the update?",
                                this.AddinName, MessageBoxButtons.YesNo,
                                MessageBoxIcon.Question) == DialogResult.Yes)
            {
                string ieFullPath =
                    Path.Combine(Environment.GetFolderPath(
                                Environment.SpecialFolder.ProgramFiles),
                                "Internet Explorer\iexplore.exe");
                this.CreateProcess("\" + ieFullPath
                                + "\" \" + updateUrl + "\")
            }
        }
    }
}
The code above is based on three methods.

- **IsMSINetworkDeployed** – returns True if the application was installed via ClickTwice :).
- **IsMSIUdpdatable** – returns True if the user is permitted to update the application. In Vista or Windows 7, it is always True. If the application was installed for all users and the current user is not an administrator, the UAC popup will ask for administrator credentials.
- **CheckForMSIUppdates** – returns an empty string if there are no updates in the location specified in the Installation URL field of the Publish dialog. If a new version of the add-in is available, CheckForMSIUppdates returns one of the following strings: a URL or UNC path (the URL can be a link to setup.exe or the application downloader), or the value specified in the Download page for updates in the Preferences dialog (see Preferences).

In the code above, we use CreateProcess to launch the executable specified in the update URL. The code runs Internet Explorer in the same integrity level as the host application. Of course, you can use your code to initiate updates on the target machine. E.g., you can use the Process class to launch the default browser providing it with the URL returned by CheckForMSIUppdate.

**Customizing ClickTwice installations**

You can download Web Authoring Example (ClickTwice). This is a C# project demonstrating the use of advanced ClickTwice features. If you write in VB.NET, you can use any C# to VB.NET code converter available on the web.

**Step-by-step Samples**

Please check the following articles:

- Deploying an Office extension via ClickTwice :)
- Updating an Office extension via ClickTwice :(
## Deployment Step-by-steps

The table below contains links to corresponding step-by-step instructions describing installing and updating an Office extension.

<table>
<thead>
<tr>
<th>How you install the Office extension</th>
<th>A per-user COM add-in, RTD server, Smart tag, or Excel UDF</th>
<th>A per-machine COM add-in or RTD server</th>
</tr>
</thead>
<tbody>
<tr>
<td>A user runs the installer from a CD/DVD, hard disk or local network location</td>
<td><a href="#">Windows Installer</a> <a href="#">ClickOnce</a> <a href="#">ClickTwice :)</a></td>
<td><a href="#">Windows Installer</a> <a href="#">ClickTwice :)</a></td>
</tr>
<tr>
<td>A corporate admin uses Group Policy to install your Office extension for a specific group of users in the corporate network; the installation and registration occurs when a user logs on to the domain.</td>
<td><a href="#">Windows Installer</a></td>
<td>N/A</td>
</tr>
<tr>
<td>A user runs the installer by navigating to a web location or by clicking a link.</td>
<td><a href="#">ClickOnce</a> <a href="#">ClickTwice :)</a></td>
<td><a href="#">ClickTwice :)</a></td>
</tr>
</tbody>
</table>
Deploying a per-user Office extension via an MSI installer

Step 1. Set `RegisterForAllUsers = false`

If you develop a per-user COM add-in or an RTD server, set the `RegisterForAllUsers` property of the add-in module or RTD server module to `False`, otherwise go to Step 2. Build your project. To access the `RegisterForAllUsers` property in the Properties window, click the designer surface of the add-in module or RTD server module:

Before you modify the `RegisterForAllUsers` property, you must unregister the add-in project on your development PC and make sure that adxloader.dll.manifest is writable.
Step 2. Build your project

If you want to support Office 2010-2013 32-bit and 64-bit applications, set the **Platform target** property to **Any CPU** before building your project.

If you use a 32-bit component in your Office extension (say a native-code DLL, ActiveX DLL, or .NET assembly), you should compile with the "x86" target platform. However, please keep in mind that such an Office extension will work in Office 2000 - 2013 (32-bit) only and will not work in 64bit Office 2010-2013.

Similarly, if you use any 64-bit third-party components, you should compile with "x64" but your Office extension will work in 64bit Office 2010-2013 only.

Summing up, if you use a bitness-aware component, your extension will work for Office versions of that bitness only.

Step 3. Create a setup project

Add-in Express provides the setup project wizard accessible via **Project | Create Setup Project** menu in Visual Studio as well as via the context menu of the project item in the Solution Explorer window (shown below).
In the New Setup Project Wizard dialog fill in all the necessary fields (Title, Description, Product name and Company) and click the Next button.

On the next step, you can choose the localization of the installer UI, the file name and output directory of your setup project.
Click the Finish button. This creates a new setup project.

Step 4. Check the DefaultLocation property

Select your setup project in the Solution Explorer window and open the File System Editor. Select the Application Folder node and check the DefaultLocation property. By default, the setup wizard sets the DefaultLocation property to the user application data folder as follows:

```
[AppDataFolder][Manufacturer]\[ProductName]
```
Step 5. Check custom actions

Select your setup project in the Solution Explorer window and open the Custom Actions Editor. The following custom actions should be present in your setup project:

- Install:
  `adxregistrator.exe /install=" {Assembly name}.dll" /privileges=user`
- Rollback:
  `adxregistrator.exe /uninstall=" {Assembly name}.dll" /privileges=user`
- Uninstall:
  `adxregistrator.exe /uninstall=" {Assembly name}.dll" /privileges=user`

where `{Assembly name}` is the assembly name of your Office extension, such as COM add-in, RTD server, Smart tag, XLL, or Excel Automation add-in.

If any of the custom actions is missing, you need to add it. Pay attention to the `/privileges` command line switch, its value should be set to `user`. 
Step 6. Set PostBuildEvent

Select your setup project in the Solution Explorer window and edit the PostBuildEvent property as follows:

```
"{Add-in Express}\Bin\adxpatch.exe" "$(BuiltOuputPath)" /UAC=Off
/RunActionsAsInvoker=true
```

where `{Add-in Express}` is the full path to the installation folder of Add-in Express.

This executable patches the generated .MSI in the following ways:

- it hides the For Me and For Everyone choice in the installer UI. Hiding these options is required because the installer will fail if the user running the installer doesn't have permissions to install for all users on the PC.
- it turns off the dialog asking for administrative privileges; the UAC dialog pops up when a non-admin user runs a setup.exe on Vista/Windows 7/Windows 2008 Server. Switching off the dialog is required because entering the admin credentials will install the Office extension for the administrator and not for the current user.

The option `/RunActionsAsInvoker=true` specifies that the installer will be run with the privileges of the user who started the installer and not with the system privileges.
Step 7. Add prerequisites (optional)

Open your setup project properties (menu Project / Properties) and click the Prerequisites button. This opens the Prerequisites dialog:

If you add any prerequisites to your setup project and the *Create setup program to install prerequisite components* option in the Prerequisites dialog is checked, Visual Studio will generate the `setup.exe` (bootstrapper) file, which will comprise all information about the prerequisites. Running the setup.exe is essential for the prerequisites to be installed. But see Step 9. Running the installer below.

Step 8. Build the setup project

Build your setup project and deliver all generated files to the target PC.
Step 9. Running the installer

Please keep in mind that installation / uninstallation of an Office extension requires closing the host application.

To run the installation on the PC, you need to choose whether to run the .MSI or setup.exe. Let's consider both options.

When the setup.exe is launched, it checks whether the prerequisites are already installed. If a prerequisite is missing, the bootstrapper installs that component. If all the prerequisites are already installed, the .MSI installer launches.

When the .MSI is launched, the extension will be installed but might not run if any prerequisite is missing.

If you deploy prerequisites requiring administrative permissions, the end user will get an UAC dialog asking for administrator credentials. But entering the administrator credentials will install your Office extension for the administrator and not for the current user. Because it is impossible to impersonate the user running the installer after admin credentials are provided, this makes combinations of per-user Office extensions and prerequisites almost useless.

Step 10. Installing a new version of the Office extension

You need to change the assembly version of your Office extension as well as the version of the setup project and rebuild it. The user needs to uninstall the previous version before installing the new one.

Don't change the Product code property of your setup project. By default, when you change the Version property of your setup project, Visual Studio shows a dialog recommending that you change the Product code. Click No or Cancel in this dialog because if you change the Product code, you will get a new Office extension product and your old extension version may be not correctly uninstalled and updated when a user launches the new version installation.
Deploying a per-machine Office extension via an MSI installer

Step 1. Set RegisterForAllUsers = true

If you deploy an add-in or an RTD server on the per-machine basis, set the `RegisterForAllUsers` property of your add-in module or RTD server module to `True`. To access the `RegisterForAllUsers` property in the Properties window, click the designer surface of the add-in module or RTD server module:

Before you modify the `RegisterForAllUsers` property, you must unregister the add-in on your PC and make sure that `adxloader.dll.manifest` is writable.

All other project types (Smart tags, XLL add-ins, Excel Automation add-ins) can be installed on the per-user basis only, please see Deploying a per-user Office extension via an MSI installer.
Step 2. Build your project

If you want to support Office 2010-2013 applications 32-bit and 64-bit, set the Platform target property to "Any CPU" before building your project.

If you use a 32-bit component in your Office extension (say a native-code DLL, ActiveX DLL, or .NET assembly), you should compile with the "x86" target platform. However, please keep in mind that such an Office extension will work in Office 2000 - 2013 (32-bit) only and will not work in 64bit Office 2010-2013.

By analogy, if you use any 64-bit third-party components, you should compile with "x64" but your Office extension will work in 64bit Office 2010-2013 only.

Summing up, if you use a bitness-aware component, your extension will work for Office versions of that bitness only.

Step 3. Create a setup project.

Add-in Express provides the setup project wizard accessible via Project / Create Setup Project menu in Visual Studio as well as via the context menu of the project item in the Solution Explorer window (shown below).
In the *New Setup Project* dialog fill in all fields (*Title*, *Description*, *Product name* and *Company*) and click the *Next* button.

You can choose the localization of the installer UI, the file name and output directory.
Click the Finish button. This creates a new setup project.

Step 4. Check the DefaultLocation property

Select your setup project in the Solution Explorer window and open the File System Editor. Select the Application Folder item and make sure that the DefaultLocation property refers to a folder accessible by all users on the PC.

By default, the setup wizard sets the DefaultLocation property to the Program Files folder as follows:

```
[ProgramFilesFolder][Manufacturer][ProductName]
```

Even if your COM add-in or RTD server is registered for all users on the machine, it may not start for users that have no permissions to access the folder where it is installed.
Deploying a per-machine Office extension via an MSI installer

Step 5. Check custom actions

Select your setup project in the Solution Explorer window and open the Custom Actions Editor. The following custom actions must be present in your setup project:

- Install: `adxregistrator.exe /install="{Assembly name}.dll" /privileges=admin`
- Rollback: `adxregistrator.exe /uninstall="{Assembly name}.dll" /privileges=admin`
- Uninstall: `adxregistrator.exe /uninstall="{Assembly name}.dll" /privileges=admin`

where `{Assembly name}` is the assembly name of your add-in or RTD server.
If any of the above-mentioned custom actions is missing in the Custom actions editor, you need to add it. Please pay attention to the /privileges command line switch, its value should be set to "admin" (without quotation marks).

**Step 6. Set PostBuildEvent**

Select your setup project in the Solution Explorer window and edit the *PostBuildEvent* property as follows:

```
"{Add-in Express}\Bin\adxpatch.exe" "$(BuiltOuputPath)" /UAC=On /RunActionsAsInvoker=true
```

*Add-in Express* above is the full path to the installation folder of Add-in Express. The option /UAC=On indicates that the UAC popup will be triggered when the installer is run and UAC is enabled. The option /RunActionsAsInvoker=true specifies that the installer will be run with the privileges of the user who started the installer and not with the system privileges.

The executable specified in *PostBuildEvent* modifies the generated .MSI in order to hide the "For Me" and "For Everyone" choice in the installer and to turn on the dialog asking for administrative privileges that appears when a non-admin user runs the setup.exe on Vista/Windows 7/Windows 2008 Server with UAC on. Hiding the "For Me" and "For Everyone" options is required because using them in a wrong way may result in a run-time error if the privileges of the user running the installer don't allow installation for all users on the PC. Turning on the dialog asking for elevating permissions is necessary to register your Office extension for all users.
Step 7. Add prerequisites (optional)

Open your setup project properties (menu Project | Properties) and click the Prerequisites button. This opens the Prerequisites dialog:

If you add any prerequisites to your setup project and the Create setup program to install prerequisite components option in the Prerequisites dialog is checked, Visual Studio will generate the setup.exe (bootstrapper) file, which includes all information about the prerequisites. Running the setup.exe is essential for installing the prerequisites.

Step 8. Build the setup project

Build your setup project and deliver all generated files to the target PC.
Step 9. Running the installer

The user should have **administrative permissions** and run the `setup.exe` (not `.MSI`).

Running the `setup.exe` ensures that:

- the installer process will be run with elevated permissions on UAC-enabled systems; elevated permissions are required to register your COM add-in or RTD server for all users on the PC
- the prerequisites you selected for your project will be installed before your Office extension is installed

If you run the `.MSI`, you'll get one of the following results:

- for the user with standard user permissions - the prerequisites will not install, your extension won't be registered because administrator privileges are required
- for the user with administrative permissions - the extension will be installed but it might not run if any of the prerequisites was not previously installed

Step 10. Installing a new version of the Office extension

You need to change the assembly version of your Office extension as well as the version of the setup project and rebuild the setup project. The user needs to uninstall the previous version before installing the new one.

Don't change the *Product code* property of your setup project. By default, when you change the *Version* property of your setup project, Visual Studio opens a dialog prompting to change the *Product code*. Click *No* or *Cancel* in this dialog because if you change the *Product code*, you will get a new Office extension installer, consequently the previous version of your extension may not uninstall correctly when the user launches the new version installation.
Deploying a per-user Office extension via Group Policy

Step 1. Set RegisterForAllUsers = false

If you develop a COM add-in or an RTD server, set the `RegisterForAllUsers` property of your add-in module or RTD server module to `False`, otherwise go to Step 2. Build your project. To access the `RegisterForAllUsers` property in the Properties window, click the designer surface of the add-in module or RTD server module:

Before you modify the `RegisterForAllUsers` property, you must unregister the add-in on your PC and make sure that `adxloader.dll.manifest` is writable.
Step 2. Build your project

If you want to support Office 2010-2013 32-bit and 64-bit applications, set the Platform target property to Any CPU before building your project.

If you use a 32-bit component in your Office extension (say a native-code DLL, ActiveX DLL, or .NET assembly), you should compile with the “x86” target platform. However, please keep in mind that such an Office extension will work in Office 2000 - 2013 (32-bit) only and will not work in 64bit Office 2010-2013.

By analogy, if you use any 64-bit third-party components, you should compile with “x64” but your Office extension will work in 64bit Office 2010-2013 only.

Summing up, if you use a bitness-aware component, your extension will work for Office versions of that bitness only.

Step 3. Create a setup project

Add-in Express provides the setup project wizard accessible via Project | Create Setup Project menu in Visual Studio as well as via the context menu of the project item in the Solution Explorer window (shown below).
In the *New Setup Project* dialog fill all the fields (*Title, Description, Product name, Company*) and click the *Next* button.

You can choose the localization of the installer UI, output directory and file name.
Click the Finish button. This creates a new setup project.

Step 4. Check the DefaultLocation property

Select your setup project in the Solution Explorer window and open the File System Editor. Select the Application Folder node and check the DefaultLocation property. By default, the setup wizard sets the DefaultLocation property to the user's application data folder as follows:

```
[AppDataFolder][Manufacturer][ProductName]
```
Step 5. Delete custom actions

Select your setup project in the Solution Explorer window and open the Custom Actions Editor. If you used the Add-in Express wizard, then in the setup project that was created by the wizard there are three custom actions in which `adxregistrator.exe` is used. Delete these custom actions.

Step 6. Add new custom actions

In the Custom Actions Editor, add three new custom actions under Install, Rollback and Uninstall by selecting `adxloader.dll` as a custom action. Then change the `EntryPoint` property for each of the actions as show below:

Install: `EntryPoint = "DllRegister"`

Rollback: `EntryPoint = "DllUnregister"`
Uninstall: EntryPoint = "DllUnregister"

Step 7. Set PostBuildEvent

Select your setup project in the Solution Explorer window and edit the PostBuildEvent property as follows:

```
"{Add-in Express}\Bin\adxpatch.exe" "$(BuiltOuputPath)" /UAC=Off
/RunActionsAsInvoker=true
```

where {Add-in Express} is the full path to the installation folder of Add-in Express.
This executable patches the generated .MSI in the following ways:

- it hides the “For Me” and “For Everyone” choice in the installer UI.
- it turns off the dialog asking for administrative privileges; the UAC dialog pops up when a non-admin user runs a setup.exe on Vista/Windows 7/Windows 2008 Server.

The option `/RunActionsAsInvoker=true` specifies that the installer will be run with the privileges of the user who started the installer and not with the system privileges.

**Step 8. Build the setup project**

Build your setup project and deliver the generated .msi file to the administrator.

**Step 9. Running the installer using Group Policy**

Use Group Policy to deploy and automatically install / uninstall your product. You can read how to do it in this article: HowTo: Install a COM add-in automatically using Windows Server Group Policy.

**Step 10. Installing a new version of the Office extension**

You need to change the assembly version of your Office extension as well as the version of the setup project and rebuild your setup project. The admin needs to remove the installation package and assign a new one as described in HowTo: Install a COM add-in automatically using Windows Server Group Policy.
Deploying a per-user Office extension via ClickOnce

Step 1. Set RegisterForAllUsers = false

If you develop a COM add-in or an RTD server, set the RegisterForAllUsers property of your add-in module or RTD server module to False, otherwise go to Step 2. Fill the Assembly information. To access the RegisterForAllUsers property in the Properties window, click the designer surface of the add-in module or RTD server module:

Before you modify the RegisterForAllUsers property, you must unregister the add-in on your PC and make sure that adxloader.dll.manifest is writable.

Step 2. Fill the Assembly information

Fill in the obligatory fields of the Assembly Information as shown in the screenshot below:
Step 3. Build your project

If you want to support 32-bit and 64-bit applications of Office 2010, set the **Platform target** property to **Any CPU** before building your project.
If you use a 32-bit component in your Office extension (say a native-code DLL, ActiveX DLL, or .NET assembly), you should compile with the “x86” target platform. However, please keep in mind that such an Office extension will work in Office 2000 - 2013 (32-bit) only and will not work in 64bit Office 2010-2013.

By analogy, if you use any 64-bit third-party components, you should compile with “x64” but your Office extension will work in 64bit Office 2010-2013 only.

Summing up, if you use a bitness-aware component, your extension will work for Office versions of that bitness only.

Step 4. Open the Publish dialog

Select your project in the Solution Explorer window and choose Publish ADX Project in the project context menu.

The steps below highlights publishing your add-in in the Visual Studio user interface. All the steps can be performed using the command line utility \{Add-in Express\}\Bin\adxpublisher.exe, see the Add-in Express installation folder.

In the Publish dialog, switch to the ClickOnce deployment tab.
Deploying a per-user Office extension via ClickOnce

![Deploying a per-user Office extension via ClickOnce](image.png)
Step 5. Populate files

Click the Populate button. This creates the Publish\{AssemblyVersion\} folder and copies all files and dependencies of the Office extension (as well as the Add-in Express loader and its manifest) into that folder.

Step 6. Add additional files (optional)

If you want to add additional files and/or folders, copy the files and folders required by your project to the Publish/<AssemblyVersion> folder and click the Populate button again.
Step 7. Set application icon (optional)

You can add a .ico file and mark it as Icon File in the Type column of the Files list box. This icon will be shown in the ClickOnce installer window and Windows Start menu.
Step 8. Set the "Provider URL"

In the Provider URL field, enter the location of the deployment manifest using one of the following formats:

- web site: http://<www.website.com>/<deployment manifest name>.application
- a Virtual Directory: http://localhost/<deployment manifest name>.application
- an FTP server: ftp://<ftp.domain.com>/<deployment manifest name>.application
- a file path: C:\<folder>\<deployment manifest name>.application
- a UNC path: \\<server>\<deployment manifest name>.application

<deployment manifest name> must be in the lower case.

Step 9. Sign the installer files

Browse for the existing certificate file (.pfx or .p12) or click New to create a new one. Enter the password of the certificate if there is any.

Step 10. Set preferences (optional)

Click the Preferences button to open the following dialog window:

In this dialog, you can allow showing your custom or the built-in UI and specify the Support Location; this will show additional information about your product in the Add or Remove Programs dialog (called Programs and Features since Vista).
Step 11. Prerequisites (optional)
Click the *Prerequisites* button to specify the prerequisites of your Office extension.
Step 12. Click the Publish button

Upon clicking the Publish button the wizard generates (updates) the manifests:

- Deployment manifest - `<SolutionFolder>/Publish/<projectname>.application`
- Application manifest - `<SolutionFolder>/Publish/<AssemblyVersion>/<ProjectName>.exe.manifest`

Step 13. Upload files and folders

Upload the following files and folders to the location you specified in the Provider URL field:

- files and folders of the `Publish/{AssemblyVersion}` folder;
- the deployment manifest `{projectname}.application`;
- `setup.exe`.

For testing purposes, you can just double-click the deployment manifest in Windows Explorer.
Step 14. Running the installer

To run the installer, the user needs to open the deployment manifest (\texttt{<projectname>.application}) in Internet Explorer or Windows Explorer.

In the \textit{Application Install – Security Warning} dialog you click the \textit{Install} button.

Clicking the \textit{Install} button opens the ClickOnce installer window (see the screenshot below):
Deploying a per-user Office extension via ClickOnce

Users will see this dialog when the installation is completed successfully:

Please note that ClickOnce doesn't provide any opportunity to customize or hide dialogs and messages shown while the user is installing or updating your add-in.
Step 15. Register/unregister or uninstall the Office extension

To unregister or re-register your Office extension, the user browses for the deployment manifest (<projectname>.application) in Internet Explorer or Windows Explorer and runs it. This opens the dialog below, where the user can click either Unregister or Register.

You can add the ClickOnce module to your project to show a custom dialog instead of the above dialog.

To uninstall your product, the user goes to Control Panel -> Add or Remove Programs (Programs and Features since Vista).

Step 16. Installing a new version of the Office extension

Please see Updating a per-user Office extension via ClickOnce.
Updating a per-user Office extension via ClickOnce

Every Add-in Express module provides the \textit{IsNetworkDeployed} method, which returns \textit{True} if your Office extension was installed via ClickOnce.

Then, you can use the \textit{CheckForUpdates} method, which initiates the updating process if a new version of your Office extension is available in the location specified in the \textit{Provider URL} field. If there is an update, \textit{CheckForUpdates} opens the \textit{Update Available} dialog:

Below you will find the steps required for deploying a new version of your Office extension.

\textbf{Step 1. Increase the assembly version number}

In the \textit{AssemblyInfo}, increase the assembly version number of your project.

\textbf{Step 2. Build your project}

Just build the Office extension project.
Step 3. Open the publish dialog

Select your project in the Solution Explorer window and choose Publish ADX Project in the project context menu (you can also find the same menu item in the Project menu in Visual Studio).

This opens the Publish dialog. Switch to the ClickOnce deployment tab.
Updating a per-user Office extension via ClickOnce
Step 4. Select the new version

In the Version drop-down list, select the new version.

Step 5. Publish the new version

Go to Step 5, Populate files to see how to publish your new version.
Deploying an Office extension via ClickTwice :)

Step 1. Create an .MSI installer

The .MSI installer can be created by using the Visual Studio setup project as well as any third-party installers like WiX or InstallShield. You can find step-by-step instructions on creating the .MSI setup package in the following articles:

For per-user deployment, see steps 1 – 6 of Deploying a per-user Office extension via an MSI installer.

For per-machine deployment, look through steps 1 – 6 of Deploying a per-machine Office extension via an MSI installer.

Step 2. Open the Publish dialog

Select your project in the Solution Explorer window and choose Publish ADX Project in the project context menu (you can also find the same item in the Project menu of Visual Studio).

In the Publish dialog, switch to the MSI-based web deployment tab.
Deploying an Office extension via ClickTwice :)
Step 3. Browse for the .MSI

Specify the path to the .MSI installer in the Installer file field. The Publish wizard reads some general info about the .MSI and fills in the fields in the upper part of the dialog.

![Publish dialog](image)

Step 4. Set the Publishing location

By default, the Publish wizard suggests publishing your application to the MSIPublish subfolder of the project directory, see the Publishing location field. You can specify any suitable location in one of the following formats:

- To publish to a file share, enter the path using either a UNC path: `\<server>\<folder>` or a file path: `C:\<folder>`
- To publish to an FTP server, enter the path as `ftp://<ftp.domain.com>/`
Using an http server as a publishing location is not supported.

**Step 5. Set the Installation URL**

You can use the *Installation URL* field to specify a location from which users will download the Office extension installer.

In the *Installation URL* edit box, enter the installation location using either a fully qualified URL in the format `http://www.domain.com/<ApplicationName>`, or a UNC path using the format `\<server>\<ApplicationName>`.

**Step 6. Set an icon (optional)**

You can specify an icon in the *Icon file* field. The icon will be shown in the downloader window, which is displayed when the installer is downloaded from the installation location (see Step 12. *Running the installer*).

**Step 7. Sign the installation files**

Browse for the existing certificate file (.pfx or .p12) or click New to create a new one. Enter the password of the certificate (optional).

**Step 8. Preferences (optional)**

To hide the Windows installer UI during installation or uninstallation, you can check the appropriate option(s) in the *Preferences* dialog. You can also specify a web page that provides users with additional information about available updates or some other information about your product in the *Download page for updates* field.
Step 9. Prerequisites (optional)

Click the Prerequisites button and select any prerequisites in the dialog window.

Add-in Express generates a setup.exe to install the specified prerequisites. When the user runs the setup.exe, it installs the prerequisites and invokes the Add-in Express downloader program to install/update your Office extension (see Step 12. Running the installer for example).
Step 10. Click the Publish button

When you click the Publish button, the wizard creates all necessary files and folders in the location specified in the Publishing location field.

Let's dwell on the files and folders. version_info.xml contains information about all versions of your Office extension:

```xml
<?xml version="1.0" encoding="utf-8"?>
<application name="sampleaddin">
  <product language="1033">
    <version name="1.0.0" installationUrl="http://127.0.0.1/SampleAddinMSI" productCode="{1123D7C8-9AEE-4B9C-B36B-38635D7D7EBF}" updateType="bootstrapper">
      <files msi="SampleAddinSetup">
        <file>SampleAddinSetup.msi</file>
      </files>
    </product>
    <preferences>
      <showInstallUI>true</showInstallUI>
      <showUninstallUI>false</showUninstallUI>
      <webPage/>
      <webPage/>
    </preferences>
  </product>
</application>
```
All versions of your Office extension are stored in folders, which are named according to the formula `<language code>\<version number of the .MSI installer>`. For instance, version 1.0.0 of the sample add-in created for this article is stored in folder `1033\1.0.0`, where 1033 stands for the locale `English-United States`.

That folder contains the following files:

- the `.MSI` installer, `SampleAddinSetup.msi` in the screenshot above
- `<project name>.exe`, also called the downloader; `sampleaddin.exe` in the screenshot; this application downloads and runs the `.MSI` installer
- `setup.exe`, the bootstrapper that installs prerequisites from the `prerequisites` subfolder, and then downloads and runs the downloader
Step 11. Copy files and folders

Upload the files and folders from the *Publishing location* folder to the location you specified in the *Installation URL* field.

Step 12. Running the installer

You need to supply the user with a link to one of the two executables below:

- `<project name>.exe`, the downloader; if there are **no** prerequisites
- `setup.exe`, the bootstrapper; if there are **some** prerequisites

The user clicks the link or navigates to it with either an Internet browser or Windows Explorer.
Step 13. Uninstalling the Office extension

To uninstall your product, the user goes to Control Panel – Programs and Features.

Step 14. Installing a new version of the Office extension

Please see Updating an Office extension via ClickTwice :)
Updating an Office extension via ClickTwice :) 

Every Add-in Express module provides the `IsMSINetworkDeployed` method, which returns `True` if your Office extension was installed via ClickTwice :). Then you can use the `CheckForMSIUpdates` method in your project code to check if any updates are available. `CheckForMSIUpdates` returns:

- an empty string if there are no updates in the location you specified in the `Installation URL` field (Step 5. Set the Installation URL);
- a formatted string: a URL or UNC path if a new version of the Office extension is available; the URL can be a link to a `setup.exe` or downloader, see Step 10. Click the Publish button;
- the web page URL that you specified in the `Download page for updates` field of the `Preferences` dialog, see Step 8. Preferences (optional)

You can find a code sample that checks for updates in [Updating an Office Extension via ClickTwice :)](#).

**Step 1. Increase the version number**

Increase the version number in the `Version` property of your setup project.

Don't change the `Product code` property of your setup project. By default, when you change the `Version` property of your setup project, Visual Studio shows a dialog recommending that you change the `Product code`. Click `No` or `Cancel` in this dialog because if you change the `Product code`, you will get a new Office extension product and your old extension version may be not correctly uninstalled and updated when a user launches the new version installation.

**Step 2. Build your setup project**

Just build the setup project.
Step 3. Open the Publish dialog

Select your project in the Solution Explorer window and choose *Publish ADX Project* in the project context menu.

In the *Publish* dialog, switch to the *MSI-based web deployment* tab.
Step 4. Browse for the new .MSI package

Specify the new .MSI file in the Installer file field. The wizard reads some general info about the .MSI and fills in the fields in the upper part of the dialog.
Step 5. Publish the new version

Go to [Step 4. Set the Publishing location](#) to see how to publish your new version.
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`.

Below we have gathered together answers to typical questions. The answers are grouped into these categories:

- **Recommended Blogs and Videos**
- **Office UI Options for Developers**
- **Development Tips** – Office extensions is a peculiar development area. Recommended for those who have never developed Office extensions
- **COM Add-in Tips** – describes typical problems that may occur when you start your COM add-in.
- **Command Bars and Controls Tips** – commandbar-related stuff
- **Debugging and Deploying Tips** – deployment of Office extensions is another peculiar area
- **Excel UDF Tips** – a wealth of info on developing Excel user-defined functions
- **RTD Tips** – several points worth of your attention
- **Architecture Tips** – communicating to an Office extension, combining several Office extensions in one assembly, etc.
Recommended Blogs and Videos

If you think we miss a blog or video, write to us at support@add-in-express.com.

On COM Add-ins, Ribbon and CommandBar UI

Add-in Express 2010 and Office add-ins: Getting started
How to customize Fluent Office 2010 Ribbon UI
Office 2010 Backstage View and Add-in Express 2010
Why doesn’t Excel quit?
How to create a custom event when Excel calculation mode changes
MS Office toolbar and Ribbon UI images style guide
Video HowTo: Support Office command bars and Ribbon UI in one add-in project (VB.NET)
Video: Add-in Express 2010 in-place GUI designers (on an example of Outlook add-in)
Customizing built-in Office Ribbon groups – C# Excel add-in example

On Excel User-Defined Functions (UDFs)

XLL Add-ins and Add-in Express 2010
Video: Creating managed Excel UDFs – XLL based user-defined functions
Video: Developing an Excel Automation Add-in
HowTo: Create a COM add-in, XLL UDF and RTD server in one assembly
Invoking a COM add-in from an Excel XLL add-in: advanced sample

On Excel RTD Server

Video: Building an Excel RTD server
HowTo: Create a COM add-in, XLL UDF and RTD server in one assembly

An independent software vendor talks about creating a real-life RTD server A to Z (source code included):

- Building a Real Time Data for Excel, part 1
- Building a Real Time Data for Excel: Avoiding VSTO, part 2
- Building a Real Time Data for Excel: How RTD servers work, part 3
- Building a Real Time Data for Excel: Architecture, part 4
- Building a Real Time Data: Excel, multithreading and callbacks, part 5
- Building an Excel Real Time Data server: Providing easy-to-read function names, part 6
- Building a Real Time Data server for Excel: Talking to the GoogleMaps APIs, part 7
- Building a Real Time Data for Excel: Avoiding Application Domain misery, part 8
- Building a Real Time Data server: Embedding a GoogleMap page in an Excel Task Pane, part 9
- Building a Real Time Data server for Excel: Creating the Setup project, part 10
• Building a Real Time Data server: Changing the Excel RTD Throttle Interval, part 11
• Building a Real-Time Data server: Utility functions; building help, part 12

On Deployment

How to install multiple Office add-ins using one setup project
How to allow a user to choose target MS Office applications during installation
Video: Implementing a custom UI for Office add-ins with ClickOnce
Video: Create an Office shared add-in – building a custom Click Twice deployment package

On Outlook Development

Outlook Events Logger Add-in – release version
Outlook Item Events explained
Outlook Items and Folders Events explained
Outlook 2010 Solutions Module revisited
Outlook 2010 Fast Shutdown feature
How to get access to hidden Outlook items via Extended MAPI?
HowTo: Handle the Outlook ItemSend event
HowTo: Deal with the Recipients collection in Outlook
HowTo: Search in Outlook items programmatically
HowTo: Use the View.XML property to filter Outlook items
HowTo: Avoid limitations of Microsoft Outlook ItemAdd event
Outlook NewMail unleashed, part 4 – Writing your own solution
How To: Get a list of Outlook contacts
How To: Perform Send/Receive in Outlook programmatically
How To: Create a new Outlook Appointment item
How To: Create a new recurring Outlook Appointment item
How To: Retrieve Outlook calendar items using Find and FindNext methods
How To: Use Restrict method in Outlook to get calendar items
How to handle Outlook item’s Reply event: tracking Explorer.SelectionChange
How to handle Outlook item’s Reply event: tracking Inspector.Activate
How to handle Outlook item’s Reply event: replying from a context-menu
How To: Create a new recurrent Task item in Outlook
How To: Create a new distribution list item in Outlook
How to customize Outlook views programmatically
Advanced search in Outlook programmatically: C#, VB.NET

On Excel Development

Fast Excel add-in. Checking incoming data in XLL
Thread-safe XLL. How to get the caller address
How to add sparklines and charts to MS Excel programmatically
How to add PivotTables and Slicers to MS Excel programmatically
Fast Excel add-in. Reading and updating cells.
Excel shapes events: getting notified about user actions
How to use Evaluate to invoke an Excel UDF programmatically
How to check programmatically if the user is editing an Excel cell

On Advanced Regions and Advanced Task Panes

Start with Advanced Outlook Regions: HOWTO Samples and Advanced Excel Task Panes: HOWTO Samples.

Video: Create an advanced Outlook region and interact with Outlook object model
HowTo: Display your own header for an advanced Outlook region
Advanced region events: how to know that the user expanded your form?
How to highlight Outlook regions
HowTo: Replace the standard Outlook Task UI
HowTo: Drag and drop onto a minimized Outlook Region and Word Task Pane
HowTo: Get properties of an Outlook email item drag-and-dropped onto a .NET form
Outlook, custom task pane and drag-drop problem
HowTo: Identify the state and location of an Outlook form
Office UI Options for Developers

Installed Add-ins

In the list of installed add-ins, you can select an add-in to get additional details about it. The add-ins are categorized as follows:

- **Active Application Add-ins** – Lists the extensions that are registered and currently running in the Office application.
- **Inactive Application Add-ins** – Lists the add-ins that are present on your computer but are not currently loaded. For example, smart tags or XML Schemas are active only when the document that references them is open. Another example is the COM add-ins listed in the COM Add-ins dialog box. If the check box for a COM add-in is selected, the add-in is active. If the check box for a COM add-in is cleared, the add-in is inactive.
- **Document Related Add-ins** – Lists template files that are referenced by currently open documents.
- **Disabled Application Add-ins** – Lists add-ins that were automatically disabled because they were causing Office programs to crash.

To open the list of add-ins, follow the instructions below.

Office 2000-2003:

- The list of add-ins is not available. For the list of COM add-ins see COM Add-ins dialog.

Excel 2007, Word 2007, PowerPoint 2007:

- Click the Microsoft Office Button
- Go to **{application name}** Options->Add-ins

Outlook 2007:

- Go to the Outlook Menu -> **Tools** -> **Trust Center** -> **Add-ins**

Office 2010-2013:

- Click **File** -> **Options** -> **Add-ins**

COM Add-ins dialog

You use this dialog box to turn an add-in on/off. You can also select an add-in to get additional information about the add-in. To open the dialog box, follow the instructions below.
In Office 2000-2003, the COM Add-ins dialog shows only add-ins registered in HKCU. In Office 2007-2013, HKLM-registered add-ins are shown too. See also Registry Keys.

Office 2000-2003:

- Go to the main menu -> Tools -> COM Add-ins

If you don't see the COM Add-ins menu item, go to the main menu -> Tools -> Customize... In the Customize dialog box, select the Commands tab. Click on the Tools entry in the Categories column, then find the COM Add-ins menu item in the Commands column and drag it on a toolbar or the main menu.

Excel 2007, Word 2007, PowerPoint 2007:

- Click the Microsoft Office Button
- Go to (application name) Options->Add-ins
- Select COM Add-ins from the Manage drop-down list at the bottom of the window and click Go.

Outlook 2007:

- Go to the Outlook menu -> Tools -> Trust Center -> Add-ins
- Select COM Add-ins from the Manage drop-down list at the bottom of the window and click Go.

Office 2010-2013:

- Click File -> Options -> Add-ins
- Select COM Add-ins from the Manage drop-down list at the bottom of the window and click Go.

Disabled Items dialog

You use the Disabled Items list to re-enable your add-in if it was disabled. An add-in is automatically disabled if it is causing Office programs to crash. To re-enable the add-in, select it and click the Enable button. To open the dialog box, follow the instructions below.

Office 2000:

- There's no such list in Office 2000.
Office 2002-2003:

- Go to the main menu -> Help -> About -> click Disabled Items

Excel 2007, Word 2007, PowerPoint 2007:

- Click the Microsoft Office Button
- Go to {application name} Options->Add-ins
- Select Disabled Items from the Manage drop-down list at the bottom of the window and click Go.

Outlook 2007:

- Go to the Outlook menu -> Tools -> Trust Center -> Add-ins
- Select Disabled Items from the Manage drop-down list at the bottom of the window and click Go.

Office 2010-2013:

- Click File -> Options -> Add-ins
- Select Disabled Items from the Manage drop-down list at the bottom of the window and click Go.

Excel Add-ins dialog

You use this dialog box to turn an Excel add-in on/off. You can also select an add-in to get additional details about the add-in. To open the dialog box, follow the instructions below.

Excel 2000-2003:

- Go to the main menu -> Tools -> Add-ins

Excel 2007:

- Click the Microsoft Office Button
- Go to Excel Options->Add-ins
- Select Excel Add-ins from the Manage drop-down list at the bottom of the window and click Go.

Excel 2010-2013:

- Click File -> Options -> Add-ins
- Select Excel Add-ins from the Manage drop-down list at the bottom of the window and click Go.
Get Informed about Errors in Ribbon markup

By default, if an add-in attempts to manipulate the user interface (UI) of Office 2007-2013 and fails, no error message is displayed. However, you can configure Office applications to display messages for errors that relate to the UI. You can use these messages to help determine why a custom Ribbon does not appear, or why a Ribbon appears but no controls show up.

Outlook 2007:

- Go to the Outlook menu -> Tools -> Options -> go to the Other tab and click Advanced Options.
- In the Advanced Options dialog box, select Show add-in user interface errors and then click OK.

Office 2007 (except for Outlook, see above):

- Click the Microsoft Office Button
- Go to {application name} Options -> Advanced
- In the General section of the details pane, select Show add-in user interface errors and then click OK.

Outlook 2010-2013:

- Click File -> Options -> Advanced
- In the Developer section of the details pane, select Show add-in user interface errors and then click OK.

Office 2010-2013 (except for Outlook, see above):

- Click File -> Options -> Advanced
- In the General section of the details pane, select Show add-in user interface errors and then click OK.

Disable all Application Add-ins

If this check box is selected, this disables all application add-ins in the application.

Project 2007:

- This option is not available in Project 2007

Outlook 2007:

- Go to the Outlook menu -> Tools -> Trust Center.
- In the Trust Center dialog box, go to the Macro Security section and note if the option No warnings and disable all macros is selected.
• Then go to the Add-ins section and note if the Apply macro security settings to installed add-ins is selected.
• If both options are selected, this effectively disables all application add-ins.

Visio 2007:

• Go to the Visio menu -> Tools -> Trust Center.
• In the Trust Center dialog box, go to the Add-ins section and see the Disable all Application Add-ins check box.

Office 2007 (except for Outlook, Project and Visio, see above):

• Click the Microsoft Office Button
• Go to \{application name\} Options->Trust Center and click Trust Center Settings.
• In the Trust Center dialog box, go to the Add-ins section and see the Disable all Application Add-ins check box

Outlook 2010-2013:

• Click File -> Options - > Trust Center.
• In the Trust Center dialog box, go to the Macro Security section.
• Note if the option No warnings and disable all macros is selected.
• Note if the option Apply macro security settings to installed add-ins is selected.
• If both options are selected, this effectively disables all application add-ins.

Office 2010-2013 (except for Outlook, see above):

• Click File -> Options - > Trust Center.
• In the Trust Center dialog box, go to the Add-ins section and see the Disable all Application Add-ins check box

**Require application add-ins to be signed**

If the Require Application add-ins to be signed by Trusted Publisher check box is selected and your add-in is not digitally signed or the certificate is invalid/outdated, this disables your add-in.

Project 2007:

• This option is not available in Project 2007

Outlook 2007:
- Go to the Outlook menu -> Tools -> Trust Center.
- In the Trust Center dialog box, go to the Macro Security section and note if the option Warnings for signed macros; all unsigned macros are disabled is selected.
- Then go to the Add-ins section and note if the Apply macro security settings to installed add-ins is selected.
- If both options are selected, this effectively disables all unsigned application add-ins.

Visio 2007:

- Go to the Visio menu -> Tools -> Trust Center.
- In the Trust Center dialog box, go to the Add-ins section and see the Require Application add-ins to be signed by Trusted Publisher check box.

Office 2007 (except for Outlook, Project and Visio, see above):

- Click the Microsoft Office Button
- Go to \{application name\} Options->Trust Center and click Trust Center Settings.
- In the Trust Center dialog box, go to the Add-ins section and see the Require Application add-ins to be signed by Trusted Publisher check box.

Outlook 2010-2013:

- Click File -> Options - > Trust Center.
- In the Trust Center dialog box, go to the Macro Security section.
- Note if the option Warnings for signed macros; all unsigned macros are disabled is selected.
- Note if the option Apply macro security settings to installed add-ins is selected.
- If both options are selected, this effectively disables all application add-ins.

Office 2010-2013 (except for Outlook, see above):

- Click File -> Options - > Trust Center.
- In the Trust Center dialog box, go to the Add-ins section and see the Require Application add-ins to be signed by Trusted Publisher check box.
Development Tips

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.

Developing an Add-in Supporting Several Office Versions

There are two aspects of this theme:

- Supporting the CommandBar and Ribbon UI in one project

You can add both CommandBar UI Components and Office Ribbon UI Components onto the add-in module. When the add-in loads, command bar or ribbon controls will show up depending on the host application version. Find additional information in Command Bars in the Ribbon UI.

- Accessing version-specific features of an Office application

Please see Choosing Interop Assemblies.

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 is a couple of 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.


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 or above 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.

You may want to read this article on our blog where we discuss the following questions:

- 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?

**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 your 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 Get details about add-in loading). Find some background info in Insight of Add-in Express Loader.

**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 rather long citation from an article by Andrew Whitechaple.

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 run time 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?](#)
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:

```csharp
string location = Assembly.GetExecutingAssembly().CodeBase;
string fullPath = new Uri(location).LocalPath; // path including the dll
string directoryPath = Path.GetDirectoryName(fullPath); // directory path
```

Configuring an Add-in

The loader manifest (adxloader.dll.manifest) allows specifying the name of the .config file in the configFileName attribute. By default, it is set to "App.Config". Note however that adding an App.config item to your add-in project is not sufficient. Build your project and look into the output folder to find out the file name of the configuration file generated by the App.config item. Now, you need to do two things:

- specify this file name in the configFileName attribute;
- make sure your add-in installers deliver this file to the target machine.

See also Add-in Loading Process and Add-in Express Loader Manifest.

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 email.

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. As described in Wait a Little, 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. See also On using threads in managed Office extensions and HowTo: Work with threads in Microsoft Office COM add-ins on our blog.

Releasing COM Objects

Office was designed and built on COM, not .NET. This means that an add-in creates COM objects when dealing with the object model of its host application. All such objects require to be released in the COM way. This is necessary because otherwise the add-in may show a different behavior in a different environment or when the add-in's host application is run programmatically.

How to prevent leaving COM objects unreleased by your code?

How to properly release Excel COM objects puts this rule as ‘1 dot good, 2 dots bad’ and explains how to rewrite such calls. In Why doesn’t Excel quit you will find what objects are created behind the scene and why they are not released when you chain COM calls.

• Don’t use foreach loops on COM collections

Such a loop accesses the collection’s enumerator internally. The enumerator is a COM object and this is the root of the problem. Use for loops instead.

• Never release COM objects obtained through the parameters of events provided by Add-in Express.

To create a friendly environment for your add-in, Add-in Express creates COM objects and relies on their state. This is why you must not release these COM objects.

You can find a comprehensive review of typical problems (and solutions) related to releasing COM objects in Office add-ins in this article – When to release COM objects in Office add-ins?

Wait a Little

Some things are not possible to do right at the moment; say, you cannot 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 OnSendMessage event always occurs on the main thread.

A message is an integer greater than 1024. To prevent interfering with messages sent from other sources, you can register a message using the RegisterWindowMessage WinAPI function, please see the function description in MSDN (here) and use a function declaration published on PInvoke.net (here).

VB.NET:
Private Const WM_USER As Integer = &H400
Private Const MYMESSAGE As Integer = WM_USER + 1000

Private Sub AdxKeyboardShortcut1_Action(sender As Object) _
    Handles AdxKeyboardShortcut1.Action
    Me.SendMessage(MYMESSAGE)
End Sub

Private Sub AddinModule_OnSendMessage (sender As Object, e As AddinExpress.MSO.ADXSendMessageEventArgs) Handles MyBase.OnSendMessage

If e.Message = MYMESSAGE Then
    ' do your stuff here
End If
End Sub

C#:
private const int WM_USER = 0x0400;
private const int MYMESSAGE = WM_USER + 1000;
private void adxKeyboardShortcut1_Action(object sender) {
    this.SendMessage(MYMESSAGE);
}

private void AddinModule_OnSendMessage(object sender, AddinExpress.MSO.ADXSendMessageEventArgs e) {
    if (e.Message == MYMESSAGE) {
        // do your stuff here
    }
}

Other code samples are provided in these blogs:

- Invoking a COM add-in from an Excel XLL add-in: advanced sample
- Outlook NewMail unleashed: writing a working solution (C# example)
- How to handle Outlook item’s Reply event: replying from a context-menu

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

<SendMessage>

- ADXAddinModule.SendMessage
- ADX01Form.ADXPostMessage
- ADXExcelTaskPane.ADXPostMessage
- ADXWordTaskPane.ADXPostMessage
- ADXPowerPointTaskPane.ADXPostMessage

<OnSendMessage>

- ADXAddinModule.OnSendMessage
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
```

Here Microsoft describe how you can modify the ProgID:

The format of a ProgID is `<Program>.<Component>.<Version>`, separated by periods and with no spaces, as in *Word.Document.6*. The ProgID must comply with the following requirements:

1. Have no more than 39 characters.
2. Contain no punctuation (including underscores) except one or more periods.
3. Not start with a digit.
4. Be different from the class name of any OLE 1 application, including the OLE 1 version of the same application, if there is one.

Before you modify the ProgId of your add-in project, unregister the project. Failing to do this creates a mess in the registry.
COM Add-in Tips

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.

Quite often it is possible 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.

If your add-in uses .NET Framework 2.0 and adxloader.log (see Get details about add-in loading) shows a significant (>10 seconds) delay between "Creating a new application domain" and "Unwrapping the managed class" stages, there are three possible solutions:

- in the .config file of the host application(s) of your add-in, use the generatePublisherEvidence tag as described in http://support.microsoft.com/kb/936707; note that the fix mentioned in this article is included in .NET Framework 2.0 SP2;
- re-target your add-in project to .NET Framework 4.0.

If you have run into a slow-start problem with an Outlook 2013 add-in, see How to avoid "A problem was detected with an add-in".

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 are not set
            tempObj = Nothing
        End Try

    End While

    Return path

End Function
```
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.dot` files. In this case, installing the add-in raises a warning when Word tries to save `normal.dot`.
- 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 customize the Word 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...
If you use an Express edition of Visual Studio

In Visual Studio Express, you need to right-click the designer surface of the module and choose *Register Add-in Express Project* in the context menu.
Command Bars and Controls Tips

CommandBar 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 object model references, a pop-up control is a built-in or custom control on a menu bar or toolbar that displays a menu when it is 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 version 2000-2003.

ControlTag vs. Tag Property

Add-in Express identifies all its controls (command bar controls) using the ControlTag property which is mapped to 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](http://www.add-in-express.com/downloads/controls-scanner.php)).

See also Connecting to Existing Command Bars and Connecting to Existing CommandBar Controls.

**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:

```vbnet
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 2007-2013 applications (an exception is Outlook 2013)? For instance, our free Built-in Controls Scanner ([http://www.add-in-express.com/downloads/controls-scanner.php](http://www.add-in-express.com/downloads/controls-scanner.php)) reports that Outlook 2007 email 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 CommandBar Terminology), this property returns `Office.CommandBar`. Note that the same applies to controls on a pop-up; command bars returned in this way are not 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 command 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 Tips

Breakpoints are Not Hit When Debugging

This problem occurs when you debug an add-in project targeting to .NET Framework 2.0-3.5 in VS 2010 - 2013. The reason is the debugger, which doesn't know what .NET Framework version your add-in uses. To find out that info, the debugger checks the .config file of the executable, which is the host application in your case. 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. If the .config file is missing or it doesn't point to a specific .NET Framework version, the debugger uses the debugging engine of .NET Framework 4.0.

To help the debugger, you can create (or modify) the .config file(s) for the Office application(s) installed on your PC. You do this using the Host Configuration command of the COM add-in module; create an empty COM add-in project, if required. Please pay attention: if the .config file of any given Office application points to a specific .NET Framework version, that .NET Framework version will be used by all .NET-based Office extensions loaded by the Office application.

Don't Use 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.

Conflicts with Office Extensions Developed in .NET Framework 1.1

In the general case, two Office extensions based on .NET Framework 1.1 and 2.0-3.5, will not work together. That occurs because of these 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.

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. Nevertheless, all 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.

Always check the log file (see Get details about add-in loading) 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 and copy that file to the Office folder on the target. See Breakpoints are Not Hit When Debugging.

How to find if Office 64-bit is installed on the target machine

Remember that the 64-bit version of Office can be installed on 64-bit Windows only. If Outlook is installed, then the value below exists in this registry key:

```
Outlook 2010-2013:
Registry view: both 32-bit and 64-bit
Key: HKLM\SOFTWARE\Microsoft\Office\{14 or 15}.0\Outlook\InstallRoot
Value name: Bitness
That value can be "x64" or "x86"; "x64" means Outlook 64-bit is installed.
```

If Outlook is not installed, you can check the following values in the following 64-bit registry key:

```
Excel, Word, PowerPoint 2010-2013:
Registry view: 64-bit
Key: HKLM\Microsoft\Office\{14 or 15}.0\{application}\InstallRoot
Value name: Path
If that value exists, then the corresponding 64-bit application is installed.
```

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.
For All Users or For the Current User?

COM add-ins, RTD servers and Smart Tags can be registered either for the user whose permissions are used to run the installer or for all users on the machine. That's why the corresponding module types provide the `RegisterForAllUsers` property (see below). Registering for all users means writing to HKLM and therefore the user registering a per-machine extension must have administrative permissions. Accordingly, `RegisterForAllUsers = False` 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:

<table>
<thead>
<tr>
<th><code>RegisterForAllUsers</code> = True</th>
<th><code>RegisterForAllUsers</code> is missing</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ProgramFilesFolder][Manufacturer][ProductName]</td>
<td>[AppDataFolder][Manufacturer][ProductName]</td>
</tr>
</tbody>
</table>

To add components to your class, drag them from the Toolbox and use the Properties window to set their properties. To create methods and events for your class, click here to switch to code view.
User Account Control (UAC) on Vista, Windows 7 and Windows Server 2008

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 register 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, click its surface and see the `Properties` window. See also [Deploying Word Add-ins](#) and [For All Users or For the Current User?](#).

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 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 an 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 prerequisites 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 prerequisites and makes possible installing the Office extension for the standard user by running the .MSI file.

**ClickOnce Application 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.0KJI\ONNRMX23.ALM\add-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](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]
```
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.

Bypassing the `AlwaysInstallElevated` Policy

When you install a per-user Office extension on a PC on which the `AlwaysInstallElevated` policy is set, the installer will run with the system privileges, and not with the privileges of the user who launches the installer. The policy is described here.

When this occurs you find the following line in `adxregistrator.log` (see Get details about add-in registration/unregistration):

```
Process Owner: System
```

It isn't possible to bypass this policy programmatically. You need to turn it off explicitly. As the description suggests you can do this by modifying a value in HKCU.
Excel UDF Tips

UDF stands for "user-defined function". Because there are two UDF types – Excel Automation Add-in and XLL Add-in – "UDF" in this chapter means either or both of these types.

What Excel UDF Type to Choose?

Excel 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.

- Due to a bug in the 64-bit version of .NET Framework 2.0, your XLL add-ins developed in .NET Framework 2.0, 3.0 or 3.5 will crash Excel 2010-2013 64-bit; the workaround is to use .NET Framework 4.0. Excel Automation add-ins are not affected by this issue.
- 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 doesn't have a description. The description of an Automation add-in is the ProgId attribute of 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.
- On the other hand, neither functions nor their arguments can have descriptions in an Automation add-in. For using descriptions in an XLL add-in, see Step #4 – Configuring UDFs. See also My XLL Add-in Doesn't Show Descriptions.
- 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.

Automation add-ins cannot modify arbitrary cells; XLL add-ins may do this, see [Can an Excel UDF Modify Multiple Cells?](#).

**My Excel UDF Doesn't Work**

You start finding the cause from [Use the Latest version of the Loader](#).

If your UDF isn't shown in the Add-in Manager dialog, then it isn't registered – see [Registry Keys](#).

Then you need to check the log file (see [Get details about add-in loading](#)) for errors. If there are no errors but .NET Framework 1.1 is 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.

```vbnet
Imports AddinExpress.MSO
...
Public Shared Function GetParameterNames(ByVal fName As String) 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.ToString() + "=" + names
        End If
    Next
```
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.

Why Using a Timer in an XLL isn’t Recommended?

You should understand that the Excel engine processes several tasks at once to deal with user input as well as recalculation planning and performing. When the OnTime event of the XLL module occurs, Excel is guaranteed to expect your actions. If you use a Timer instead, you can execute some actions when Excel doesn't expect it and this may end with a crash.
Parameterless UDFs

If, in the Insert Function dialog, you choose a user-defined function that does not accept parameters, Excel shows the Function Arguments dialog informing the user that "This function takes no arguments".

It is obvious that creating such functions requires that you not add parameters to your UDF.

In an XLL add-in, however, you may run into a bug in the Excel API. The bug shows itself if a parameterless function is mentioned in an ADXExcelFunctionDescriptor that has a non-empty string in the Description property. If this is the case, you'll get another version of the Function Arguments dialog:

That is, to bypass that issue, you need to leave the ADXExcelFunctionDescriptor.Description property empty.

Can an Excel UDF Modify 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. With Excel 2007-2013, you can use the following workaround for this limitation: your XLL function can assign such a value by running a command-equivalent Excel 4 macro via the ExecuteExcel4Macro() method of the Excel.Application interface. Add-in Express incorporates that workaround by providing the safeMode parameter in constructors of the AddinExpress.MSO.ADXExcelRef type; if that parameter is true, calling the ADXExcelRef.SetValue() method triggers a hidden macro that Add-in Express registers specially for that situation.
Can an Excel UDF Return an Empty Cell?

The answer is "no". A formula does not put its result in a cell. Instead, the formula causes the cell to display the result of the formula. That is, the cell contains the formula, not the result; the cell just displays the result. That is, an UDF cannot return an empty cell; it must return a value and then Excel interprets the value.

Using the Excel Object Model in an XLL

At [http://support.microsoft.com/kb/301443](http://support.microsoft.com/kb/301443), they say:

A function that is defined in an XLL can be called under three circumstances:

5. During the recalculation of a workbook
6. As the result of Excel's Function Wizard being called on to help with the XLL function
7. 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. If you develop an XLL, you can use the `ADXXLLModule.IsInFunctionWizard` property, see [Step #3 – Creating a User-Defined Function](#) for a code sample.
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!):

```csharp
private bool InFunctionWizard {
    get {
        return (ExcelApp.CommandBars["Standard"].Controls[1].Enabled == false);
    }
}
```

Nevertheless, this code requires polishing since it leaves a number of COM objects non-released; please check Releasing COM Objects.

### Returning an Error Value from an Excel UDF

In Excel add-ins, you use `AddinExpress.MSO.ADXExcelError`. In Excel Automation add-ins, see `AddinExpress.MSO.ADXxlCVError`.

### XLL and Shared Add-in Support Update

If you develop an XLL in 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 prerequisites 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 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.

```csharp
// - select 3 consequent cells in a row,
```
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.

```csharp
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)
        Dim callerAddress As String = caller.ConvertToA1Style
        Dim range As Excel.Range = _Module.ExcelApp.Range(callerAddress)
        Dim oldCultureInfo As CultureInfo = Thread.CurrentThread.CurrentCulture
        range.NumberFormat = "mm/dd/yyyy"
    End If
End Function
```
```vbnet
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) = ADXICLError.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.

**Multi-threaded XLLs**

In Excel 2007 and higher, your XLL UDF can be registered as multi-threaded. This allows Excel to call your UDF simultaneously on multiple threads and in this way minimize the time required for recalculation. This is especially useful when your UDF makes a call to a server so that Excel may issue another call while the first one is being executed; in the same situation in Excel 2000-2003, the second call waits for the first one to finish.

A UDF becomes thread-safe formally if you set the `IsThreadSafe` property of the corresponding function descriptor object (see Step #4 – Configuring UDFs, for instance) to true. To be thread-safe in reality, your UDF should comply with several rules (the information below is a compilation of *Financial Applications using Excel Add-in Development in C/C++, 2nd Edition*):

- Don’t read values of an uncalculated cell (including the calling cell)
- Don’t write any values to a cell
- Use `ADXXLLModule.CallWorksheetFunction` to call functions such as `xlfGetCell`, `xlfGetWindow`, `xlfGetWorkbook`, `xlfGetWorkspace`, etc.;
- Don’t define or delete XLL-internal names by calling `xlfSetName` via `ADXXLLModule.CallWorksheetFunction`
- Use critical section when accessing thread-unsafe data such as static variables, etc.
- Don’t make calls to thread-unsafe functions
Asynchronous XLLs

You can create asynchronous user-defined functions in Excel versions starting from 2010.

An asynchronous UDF consists of two parts. One of them, which works synchronously, is quite a typical UDF. Excel calls it when you specify the UDF name in an Excel formula. Typically, it performs the following functions:

- It checks input parameters
- It returns a value to Excel if the UDF is called from the Insert Function Wizard
- In case of error, it returns an ADXExcelError or any suitable value
- It calls other XLL functions
- It initiates an asynchronous operation and supplies it with parameters

The synchronous part is declared as a method with no return value (a Sub in VB.NET). To return anything to Excel, the UDF declaration must include a parameter of the ADXExcelAsyncCallObject type; it must be the last parameter in the declaration. Here is an example:

```csharp
public static void AsyncUdf(
    object arg1,
    object arg2,
    ADXExcelAsyncCallObject asyncCallObject) {}
```

The ADXExcelAsyncCallObject class is responsible for providing Excel with the result of an asynchronous call. It also wraps a handle that Excel uses to identify any given asynchronous call. ADXExcelAsyncCallObject provides only one method:

```csharp
public bool ReturnResult(object value);
```

This method returns false if Excel rejects the value passed to it. In our experience, this is always the case when calling this method for the second (and subsequent) time during the same call of the asynchronous UDF. In other words, calling this method in a loop doesn’t make sense.

In the synchronous part of your asynchronous user-defined function, you call this method to return a value to Excel in case of error or if your UDF is called form the Insert Function wizard. In the asynchronous part, you call this method to return the calculation result or an error.

The asynchronous part of the UDF is what the synchronous part starts to perform the calculation and return the result. For instance, it can be a thread started by the BackgroundWorker class. Note that the asynchronous part must have access to the ADXExcelAsyncCallObject that the synchronous part receives from Excel. In case of the BackgroundWorker class, the synchronous part must pass the ADXExcelAsyncCallobject in the parameters of the RunWorkerAsync method.
The asynchronous part of the asynchronous UDF cannot perform any XLL-related calls!

When the asynchronous part completes or encounters an error, it invokes the ReturnResult method of the ADXExcelAsyncCallObject. Below is a schematic template of the asynchronous part when using the BackgroundWorker class:

```csharp
private void AsyncWork(object sender, DoWorkEventArgs e) {
    ADXExcelAsyncCallObject asyncCallObject =
        // retrieve the ADXExcelAsyncCallObject from e.Argument
    try {
        object result = // do some job here
        asyncCallObject.ReturnResult(result);
    } catch (Exception) {
        asyncCallObject.ReturnResult(ADXExcelError.xlErrorValue);
    }
}
```

COM Add-in, Excel UDF and AppDomain

It is 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 UDF 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 Tips

No RTD Servers in EXE
Add-in Express supports developing 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.

Inserting the RTD Function in a User-Friendly Way
The format of the RTD function 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:

```vba
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 XLCELL32.DLL. Here is a sample

```vba
Public Shared Function CurrentPrice(ByVal topic1 As String) As Object
    If _Module.IsInFunctionWizard Then Return "This UDF calls an RTD server."
    Return _Module.CallWorksheetFunction(ADXExcelWorksheetFunction.Rtd, _
        "Stock.Quote", Nothing, topic1, "Last")
End Function
```
Troubleshooting

Troubleshooting add-in registration

If the registration log file is not created/rewritten when you run the installer, check Get details about add-in registration/unregistration for the location and file name of the registration log file and make sure that the permissions of the user starting the installer allows creating file in that location.

If the file doesn't contain any warnings or errors, this means the add-in has been registered successfully.

If running the installer produces an addxregistrator.log containing warnings or errors, contact us to find the cause(s) and solution(s). Typical are these issues:

- Insufficient privileges when registering a per-machine add-in
- BadImageFormatException when registering the add-in
- Exception when registering the add-in

Insufficient privileges when registering a per-machine add-in

On Windows XP, this occurs if a standard user runs the installer of a per-machine add-in. On UAC-equipped systems, this occurs if the installer of a per-machine add-in is run with non-elevated privileges; typically, this occurs if the user starts the .MSI, not setup.exe.

Symptoms. The header of the addxregistrator.log file contains these lines:

- Command Line: {… omitted…} /privileges=admin
- Run 'As Administrator': No
- Process Elevated: No

Suggestions. Run the installer with administrative permissions. On UAC-equipped systems, you can bypass the situation above by running setup.exe, not the .MSI. For surety, you can start setup.exe using the "Run as administrator" command.

BadImageFormatException when registering the add-in

This exception occurs when a 32bit process loads a 64bit assembly and also when a 64bit process loads a 32bit assembly. In the context of registering an Add-in Express add-in this occurs if you build your assembly for the x64 platform; the point is the addxregistrator.exe executable (see How Your Office Extension Is Registered) is 32 bit.

Suggestion. Build your add-in for the Any CPU platform.
Exception when registering the add-in

When your add-in is registered or unregistered, Add-in Express creates an instance of the add-in module. Because in this situation the add-in isn't loaded in the host application, you cannot 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.

Troubleshooting add-in loading

Make sure that the generateLogFile attribute is set to true in the add-in manifest, see Add-in Express Loader Manifest. By default, the log file – the default file name is adxloader.log – is generated in the folder {Documents}\Add-in Express in the profile of the user who starts the host application, see Get details about add-in loading. If the manifest contains the logFileLocation tag, make sure that the user starting the host application is permitted to create files in the folder specified in this tag.

Start the host application and check if an adxloader.log file is created/rewritten in the default location or in the location specified by the logFileLocation tag (see above). If starting the host application produces an adxloader.log containing warnings or errors, contact us to find the cause(s) and solution(s).

If the file doesn’t contain any warnings or errors, this means the add-in has been loaded successfully. The record The managed add-in class has been created successfully means the constructor of the module was called and no error has occurred.

If adxloader.log isn't created, the variants are these:

- Add-in is not Registered
- Per-User Add-in is Registered for System User
- Add-in is Inactive
- Add-in is Disabled
- Digital Signature is Missing or Invalid
- Administrator cannot Load Per-user Add-in
- Per-machine Add-in Loads only for Administrator
- All Application Add-ins are Disabled by User
- All Application Add-ins are Disabled by Administrator
- XLL add-in is Blocked

One more issue is described here. Below is a citation:
... the add-in was still partially registered for "all users" in HKLM hive of registry, and this was blocking the "per user" registration from working correctly.

Finally, there was a case of Excel started using the account other than the current user's account; supposedly, this was due a security setting. The most intriguing was the fact that per-user COM add-ins were loaded correctly while XLL add-ins do not load.

Add-in is not Registered

Symptoms.

- All Office versions. The add-in registry keys are missing in the locations given in Locating COM Add-ins in the Registry.

Suggestions. Check if the adxregistrator.log file (see Get details about add-in registration/unregistration) contains any information about the problems. If you have difficulties with understanding the file and/or the problem, send the file to support@add-in-express.com.

Per-User Add-in is Registered for System User

Symptoms.

- All Office versions. The adxloader.log file (see Get details about add-in loading) is missing. The adxregistrator.log file (see Get details about add-in registration/unregistration) contains the following line:

  Process Owner: System

Suggestions. See Bypassing the AlwaysInstallElevated Policy.

Add-in is Inactive

This occurs if the user turns off the add-in or if the add-in generates an unhandled exception at startup.

Symptoms.

- Office 2007-2013: The add-in is listed under the Inactive Application Add-ins category (see Installed Add-ins) and the add-in is not checked in the COM Add-ins dialog.
- All Office versions: The add-in is not checked in the COM Add-ins dialog.
Suggestions. Enable the add-in in the COM Add-ins dialog. If this creates/rewrites an adxloader.log, then follow the instructions in Troubleshooting add-in loading. Otherwise, see Add-in is not Registered.

All Application Add-ins are Disabled by User
This occurs if the user has chosen to disable all application add-ins.

Symptoms.

- Office 2007-2013: The add-in is listed under the Active Application Add-ins category (see Installed Add-ins); the add-in is checked/ticked in the COM Add-ins dialog; clicking the add-in in the dialog displays the message "The add-in you have selected is disabled by your system administrator."

Suggestions. Uncheck/clear the Disable all Application Add-ins checkbox.

All Application Add-ins are Disabled by Administrator
This occurs if the administrator has chosen to disable all application add-ins.

Symptoms.

- Office 2013: No add-ins are listed under the Active Application Add-ins category (see Installed Add-ins); the add-in is not checked/ticked in the COM Add-ins dialog you cannot set it; clicking the add-in in the dialog displays the message "Not loaded. The user selected to disable macros."


Add-in is Disabled
Office can disable the add-in if it crashes the Office program.

Symptoms.

- Office 2007-2013: the add-in is listed under the Disabled Application Add-ins category; see Installed Add-ins.
- Office 2007-2013: the add-in is listed under the Disabled Application Add-ins category; see Installed Add-ins.
- Office 2002-2013: the add-in is listed in the Disabled Items dialog.

Suggestions. Open the Disabled Items dialog and enable the add-in. Look for the cause of the issue; debug the add-in.
Administrator cannot Load Per-user Add-in

This applies to Vista and all subsequent Windows versions. This doesn't apply to Windows 2000 and Windows XP.

Symptoms.

- On Vista with no service packs installed, Windows ignores per-user COM objects if the process is created by a member of the local Administrators group and UAC is disabled. This is by design.
- On Vista SP1 and all subsequent versions, Windows ignores per-user COM objects if UAC is enabled and the process is elevated. This is by design.

Suggestions. On Vista, enable UAC. On Vista SP1 and later, run the host application non-elevated. For instance, you may need to look for the "Run as administrator" checkbox in the shortcut starting the application.

Digital Signature is Missing or Invalid

The user requires all application add-ins to be signed and the add-in is not signed or the certificate is invalid/outdated

Symptoms.

- The add-in is not digitally signed or the certificate is invalid/outdated.
- The Require application add-ins to be signed check box is checked/ticked.

Suggestions. Obtain a valid certificate (see Introduction to Code Signing) or clear the checkbox.

XLL add-in is Blocked

An XLL add-in doesn't load because the user/administrator has blocked certain file types from being loaded in Excel.

Symptoms.

- Office 2010-2013: the Excel Add-in Files item is checked/ticked on the File | Options | Trust Center | trust Center Settings | File Block Settings page

Suggestions. Uncheck the item above, modify the registry, or contact the administrator.
Per-machine Add-in Loads only for Administrator

This occurs if an add-in is installed to a location that other users don’t have permissions for. Typically, this occurs if the default installation folder for a per-machine add-in is set to [AppDataFolder] or [LocalAppDataFolder], rather than [ProgramFilesFolder].

Symptoms:

- The add-in loads correctly for the administrator who installed it. Other users don't see the add-in.

Suggestions. Install the add-in to a location accessible by all users on the machine.

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
- PublicKeyToken of your add-in assembly doesn't correspond to the PublicKeyToken mentioned in the Add-in Express Loader Manifest. See below.

You can find the PublicKeyToken of your add-in 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.
Architecture Tips

Developing Multiple Office Extensions in the Same Project

Add-in Express supports adding several modules in a project, every module representing an Office extension. 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](#).

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 using the Add New Item dialog of your add-in project, 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:

```plaintext
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")
```
```vbnet
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:

```vbnet
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/.