Authoring .NET Installations

InstallShield

Technical White Paper

(version 4.0)

Last Updated: October 1, 2002

Abstract

With the introduction of Microsoft® Visual Studio® .NET, Microsoft has provided a set of entirely new development tools, including powerful new languages such as Microsoft Visual C#™ .NET. While much of the whirlwind of information about Visual Studio .NET and Microsoft's .NET initiative focus on things like Web Services, there has been a distinct lack of information regarding more basic issues, such as: How are .NET applications deployed easily to end-users? How does a developer package a mixture of .NET and Win32 code to create a reliable installation? How is the Microsoft .NET runtime redistributed? This paper will answer these questions, as well as explain how .NET has changed the world of software installation and dispel some of the myths involving installation of .NET applications.

Table of Contents

Terminology

The following terms will be used in this discussion of .NET installations:

.NET, .NET Code, and .NET Applications — While .NET generically refers to a large Microsoft business initiative involving web-based services, in this paper .NET Applications refer to programs that are created with Visual Studio .NET and require the .NET Framework to operate. .NET code is also known as managed code, because it is managed by the .NET Framework.

.NET Framework — Also known as the Common Language Runtime, this is a collection of redistributable files required to be present on a system for a .NET application to function properly.

.NET Assembly — A .NET Assembly is a collection of files and their related metadata. You can think of a .NET Assembly as something similar to a Windows Installer component, as .NET Assemblies are always installed and removed as a singular unit. Because their architectures are so similar, the Windows Installer is the perfect engine to install .NET Assemblies.

A .NET Assembly can come in many forms. For the purpose of this paper we will discuss a simple .NET Assembly containing only one file with the metadata embedded in the header of the file and assume that this assembly was created in Visual Studio .NET. .NET Assemblies can be installed local to the application or to the Global Assembly Cache, if appropriately signed, to share with other applications.

Win32 and Win32 Applications — Refer to traditional 32-bit Windows programs that don't require the .NET Framework to operate correctly. Win32 code is also known as unmanaged code, because it is not managed by the .NET Framework.

Inter-operable Code and COM-interop — Inter-operable code generically refers to code that contains both Win32 and .NET code. COM-interop specifically refers to the mechanism used for Win32 and .NET code to work with one another.

Isolated Applications and Side-by-Side Components — An Isolated Application is an application designed to have a "no impact" installation on the system. All components that belong to the application are installed isolated to just that application. Side-by-Side Components refer to components (such as DLL's) that are written so that different versions of the same component can be loaded into memory (side-by-side) at the same time without causing any conflicts.

How .NET Changes the World of Software Installation

The Myth of Self-Installing Code: Why You Can't Just Use XCOPY

You may have heard that you can install .NET Applications using XCOPY. You may have even seen a demonstration of how it works. Of course, this would bring up the question, "Why do I even need a software installer anymore?" While it is true that some simple .NET applications can be copied to a system so they function properly without registration, this represents an incomplete view of what actually needs to happen during the software installation process.
Software installation is more than simply getting your files from point A to B. Software installers provide a friendly, reliable, and consistent way for the end-user to place software on their machine. First, most installers manage the process of collecting input from the user regarding software installation preferences through a consistent and familiar user interface. The Installer then manages creation of a directory hierarchy, shortcut creation, upgrades, Internet-based installation, license-key management, software uninstallation, and many other tasks. In the above example, the user is faced with performing many of these tasks alone. Other tasks, such as license-key management, simply aren't possible with "self-installing" code.

Additionally, applications that are a mix of managed and unmanaged code can't be self-installed. In fact, the process of installing a mix of managed and unmanaged code is more complex than installing traditional unmanaged code. Since most developers won't be throwing out their unmanaged code and starting from scratch with managed code, many of these mixed environments are expected to exist for a long time. Finally, and most importantly, the .NET Framework must be installed and configured on the end-user's machine. Only an installation package can determine if the appropriate version of the .NET Framework is installed on the end-user's machine and install or update the .NET Framework as needed before the application is installed on the system.

Installation Requirements

BUILDING THE INSTALLATION PACKAGE

Microsoft advises that the Windows Installer Service be used to install and configure .NET applications on an end-user's system. To do this, a Windows Installer package must be configured appropriately to install the .NET application. This process involves determining the correct information to write to the Windows Installer tables, including Custom Actions for any special-cases that the Windows Installer does not handle natively, and including the appropriate version of the .NET Framework redistributable in the installation package.

INSTALLING THE .NET APPLICATION

Once the installation package is delivered to the end-user's system, a number of tasks must be completed in addition to the typical tasks associated with software installation. Firstly, both the Windows Installer Service as well as the .NET Framework need to be installed and configured on the system. Secondly, the Windows Installer Service must be called to install and configure all of the assemblies and other data associated with the .NET application. Finally, the system must be configured to allow the application to be cleanly removed when no longer needed.

Inter-Operable Code

Visual Studio .NET provides methods for developers to write managed code that works together with traditional, non-managed code. This functionality is known as COM-interop. While this functionality is extremely useful for developers, it introduces a level of complexity into software installation that was previously unknown. Specifically, to register a component for COM-interop requires a separate utility and generation of individual registry files for each component. These files then need to be installed with the inter-operable components. Additionally, tracking dependencies between managed and unmanaged code can be a daunting task, to say the least. Developers require an easier way to install inter-operable code on end-users machines.

Isolated Applications and Side-by-Side Components (Win32)

The .NET initiative even affects the installation of Win32 components. As part of the Windows XP Operating System and .NET, Microsoft has provided an infrastructure for reducing the occurrences of DLL Hell. Applications often rely on shared resources to operate correctly. The concept behind shared resources is that once a shared resource is on the system, other applications that require that resource don't need to install it to use it. DLL Hell is an unfortunately common scenario where an application may require a newer version of a shared resource than what is on the machine, so the application's installer overwrites the old resource with a newer version. Over time, newer versions of shared resources may be incompatible with older versions, and since only one version of a shared resource can exist on the machine, applications that rely on the older resource are broken.

The solution to the DLL Hell problem that Microsoft has provided allows for Isolated Applications and Side-by-Side Components. Isolated Applications are applications that install dedicated versions of all components required to run the application — these applications are then completely unaffected by changes made to the system by installation of other applications. Side-by-Side Components are shared resources that are designed to be installed and run on the system alongside older and newer versions of the same shared resource. Many shared resources that developers currently rely on are being written as Side-by-Side Components for Windows XP.

Developers can take advantage of Side-by-Side Components and partially isolate their application by creating an Application Manifest. The Application Manifest is installed with the application and expresses application dependency and version information. With this information, the operating system can be sure to always load the appropriate version of a shared resource into memory for that application. For example, an application may be written and tested on COMCTL32 v5 — when installed the application manifest will express the application's dependency on v5 of COMCTL32. Even if the operating system has other versions of COMCTL32, the application will always be run with v5, therefore avoiding DLL Hell for that application.

New .NET-related Functionality in InstallShield Developer

InstallShield Developer introduces many new views, options and wizards to assist setup developers in creating installations for .NET applications. Additionally, InstallShield Developer is the only robust installation authoring environment to offer integration with Visual Studio .NET. This section will walk you through all the new .NET-related functionality in InstallShield Developer. To review the process of building .NET installations from within the Visual Studio .NET IDE, see the section titled: Working with Visual Studio .NET and InstallShield Developer on page 10.

Enabling the .NET Functionality in InstallShield Developer

Before you begin using the .NET functionality in InstallShield Developer, it is important that you have the .NET Framework installed and configured on your machine. There are a number of ways to do this:

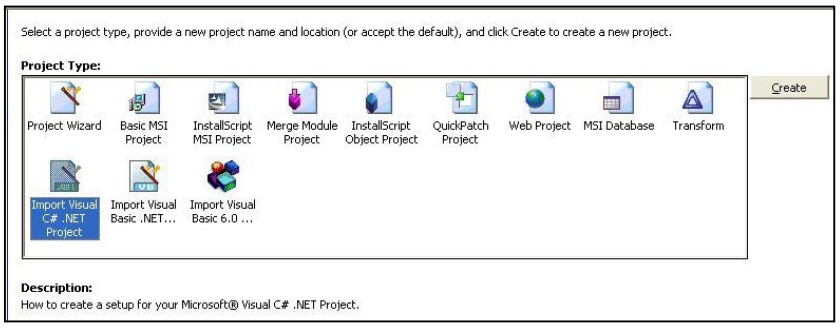
If you are using InstallShield Developer 7, download the InstallShield Developer .NET Update from http://www.installshield.com/products/dotnet.asp. This update will install and configure the .NET Framework on your machine so you can utilize all the functionality described in this paper.

Install Visual Studio .NET. Installing Visual Studio .NET will install and configure the .NET Framework as well as allow you to utilize InstallShield Developer's Visual Studio .NET integration features.

Download the Microsoft .NET Framework Software Development Kit from http://msdn.microsoft.com. This will install and configure the .NET Framework on your machine as well as give you additional documentation.

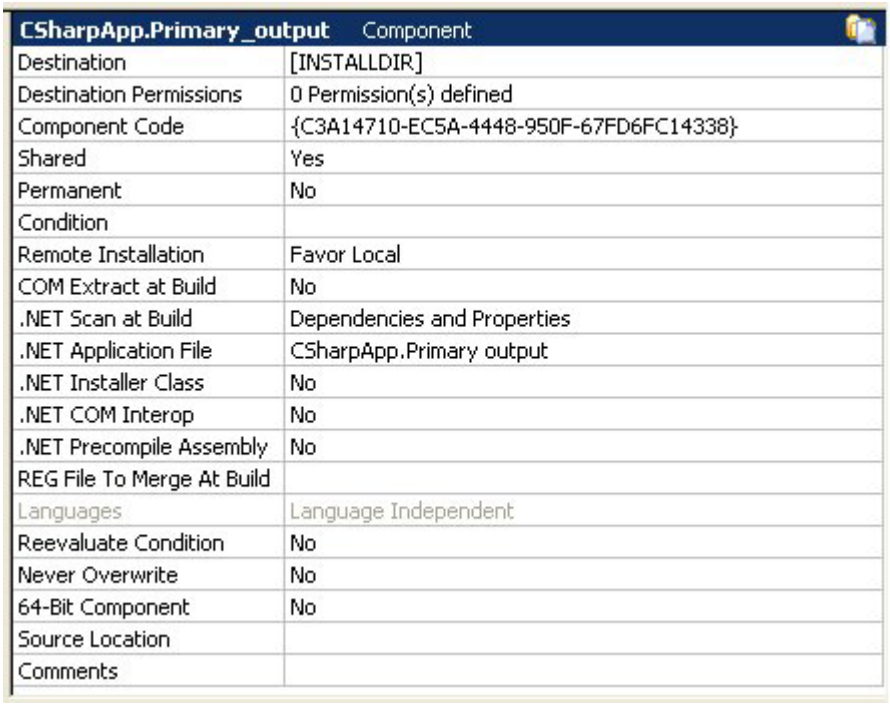
InstallShield Developer 8 will automatically install the .NET Framework on your system.

New Project Types



InstallShield Developer includes two .NET specific project types: the Visual C# .NET Project type and the Visual Basic .NET Project type. These project types provide quick ways to import your .NET application and jumpstart your installation development.

New Component Properties



CSharpApp.Primary_output	Component
Destination	[INSTALLDIR]
Destination Permissions	0 Permission(s) defined
Component Code	{C3A14710-EC5A-4448-950F-67FD6FC14338}
Shared	Yes
Permanent	No
Condition	
Remote Installation	Favor Local
COM Extract at Build	No
.NET Scan at Build	Dependencies and Properties
.NET Application File	CSharpApp.Primary output
.NET Installer Class	No
.NET COM Interop	No
.NET Precompile Assembly	No
REG File To Merge At Build	
Languages	Language Independent
Reevaluate Condition	No
Never Overwrite	No
64-Bit Component	No
Source Location	
Comments	

Components have five new .NET-related properties, which are all displayed in their property sheet: .NET Scan at Build, .NET Application File, .NET Installer Class, and .NET COM Interop. These properties provide advanced control of how InstallShield builds components that contain Assemblies. In most cases these properties can remain set to their default.

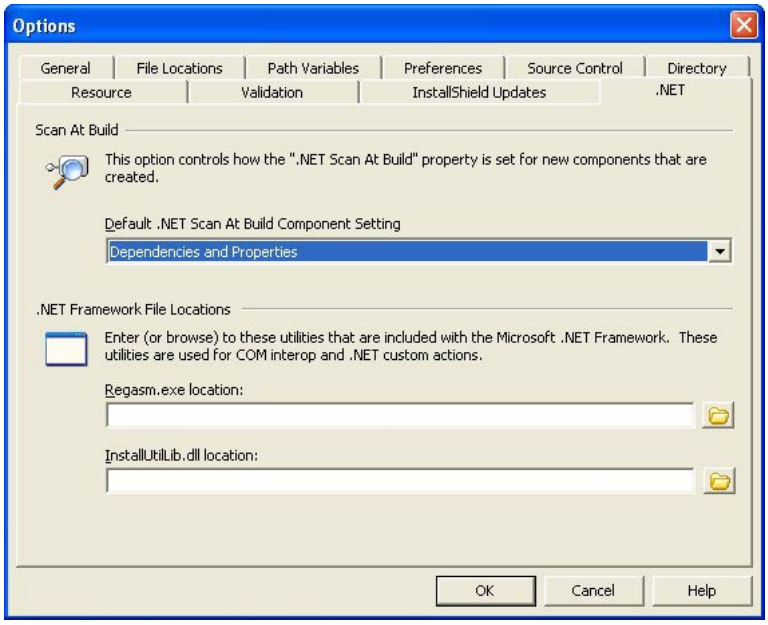
.NET SCAN AT BUILD

At build-time, InstallShield can automatically include all of a .NET Assembly's Properties and/or Dependencies in the setup package. The options available are:

- **None** — No build-time dependency or property scanning occurs. This option assumes that you are defining all properties through the Component's Advanced Settings or using the Component Wizard to perform a static scan and that you are including all dependencies manually.

- **Dependencies and Properties** — Scan for both dependencies and properties at build-time.

- **Properties Only** — Scan for properties but not dependencies at build-time. This option assumes you are including all dependencies manually.



The Options Dialog (Tools ▸ Options) has a new tab labeled **.NET**. This tab configures how the **Scan At Build** property is set by default for new components.

.NET APPLICATION FILE

This Parameter sets the **File_Application** column in the **MsiAssemby** Table, identifying the application that uses your Assembly. Normally this setting is left blank and the key file of the component is used as the Application File. If you wish to override this behavior and set a different Application File, such as the primary application executable, specify a different file here.

.NET INSTALLER CLASS

If your .NET Assembly takes advantage of the **System.Configuration.Install** namespace in order to support .NET custom actions written in the same language as your .NET application, set this property to **Yes**. This property defaults to **No**.

.NET COM INTEROP

Extra work must be done to set up a .NET Assembly for COM Interop at install time. Fortunately, InstallShield does all the extra work for you. Toggle this property to **Yes** if your Assembly requires COM Interop functionality. This property defaults to **No**.
(**Note:** This option only works if you are installing your assembly to the Global Assembly Cache.)

.NET PRECOMPILE ASSEMBLY

If your assembly takes advantage of just-in-time compilation, set this property to YES and InstallShield will automatically compile your code when installed.
(**Note:** This functionality does not work for Assemblies being installed into the GAC. It also does not work for Assemblies being upgraded by a minor upgrade.)

Additionally, a Component's Destination field now contains a new option, **[GlobalAssemblyCache]**. When this destination is selected, the component will be installed to the Global Assembly Cache.
(**Note**: This works only for appropriately signed and configured assemblies.)

New Assembly Node for Component Advanced Settings

Under a component's Advanced Settings, there is a new Assembly node. This node allows for exact control over how an Assembly is built into your package. In order to use the Assembly node, be sure that you have the component's .NET Scan-at-Build property set to **None**. Right click on the **Assembly** node and select either **New .NET Assembly** or **New Win32 Assembly**.

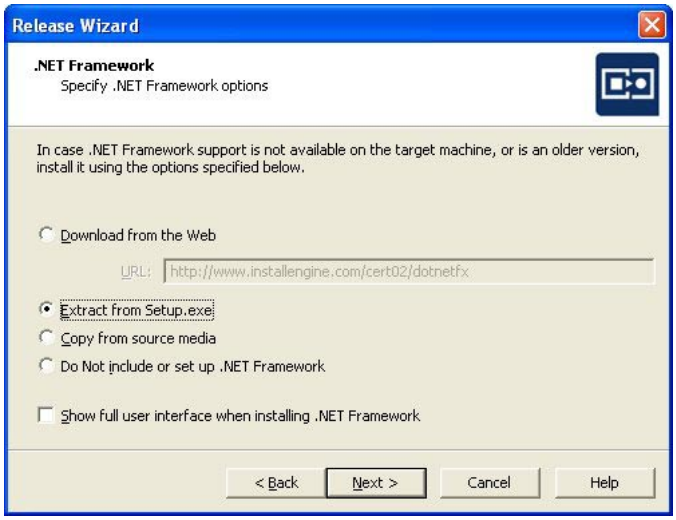
NEW .NET ASSEMBLY

Selecting this assembly type will allow you to manually configure the settings for a .NET Assembly. In most cases InstallShield's build-time .NET property extraction is sufficient and there is no need to manually specify your Assembly settings.

NEW WIN32 ASSEMBLY

Win32 Assemblies allow you to take advantage of the Application Isolation functionality that is built into Windows XP and later operating systems. InstallShield does not currently support build-time property extraction of Win32 Assemblies, so if you are working with a Win32 Assembly you will always need to define the property values manually. Be sure your manifest file is contained in the component you are working on and select the file from the drop-down for the Manifest Property. To finish the configuration of your Assembly, specify the name, type, and version properties.
(**Note**: Your Assembly may have other properties that need to be defined; if this is the case left-click in the property grid to add a new property.)

New .NET Framework Panel in the Release Wizard



In order to install a .NET Assembly on an end-user's machine, the .NET Framework must first be installed. InstallShield will automatically determine if the user's machine contains an appropriate .NET Framework for your application and will install or update the framework, if necessary. This panel of the Release Wizard controls how InstallShield performs this task.

DOWNLOAD FROM THE WEB

InstallShield will automatically download the redistributable for the .NET Framework from the specified URL, if necessary, and install it on the end-user's machine. This option requires that the end-user has an active Internet connection. If the version of the .NET Framework on the end-user's machine is appropriate for your application, no files will be downloaded.

EXTRACT FROM SETUP.EXE

The .NET Framework redistributable will be compressed into the Setup.exe file and installed, if necessary, on the end-user's machine.

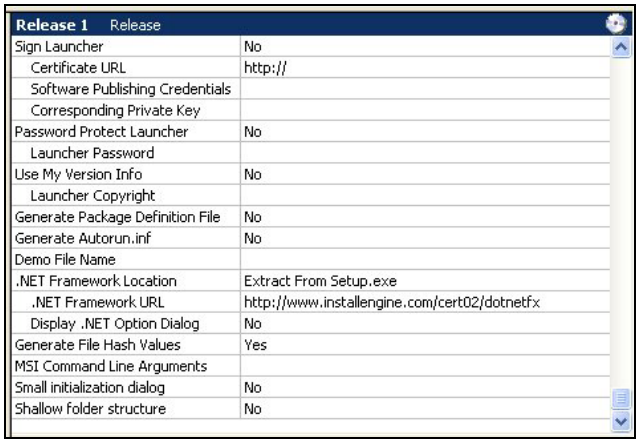
COPY FROM SOURCE MEDIA

The .NET Framework redistributable will be stored on the release media and installed, if necessary, on the end-user's machine.

DO NOT INCLUDE OR SETUP .NET FRAMEWORK

The .NET Framework will not be included with this package. InstallShield will assume that the appropriate version of the .NET Framework is already installed on the end-user's machine.

New Release Properties



The property sheet for a release now has three new .NET-related properties.

.NET FRAMEWORK LOCATION

The options in this dropdown are the same as the options available from the Release Wizard.

.NET FRAMEWORK URL

The URL where InstallShield will look for the .NET Framework redistributable, provided that **Download From The Web** is selected for the .NET Framework Location property.

DISPLAY .NET OPTION DIALOG

Setting to YES will give the user the option to not install the .NET Framework at install-time.

.NET BUILD CONFIGURATION

This property associates a specific Solution Configuration in Visual Studio .NET with this release. For example, if you specify **Debug**, then the Debug outputs from the Visual Studio project will be included at build-time. If you specify **Release**, then the Release outputs from the Visual Studio project will be included at build-time.

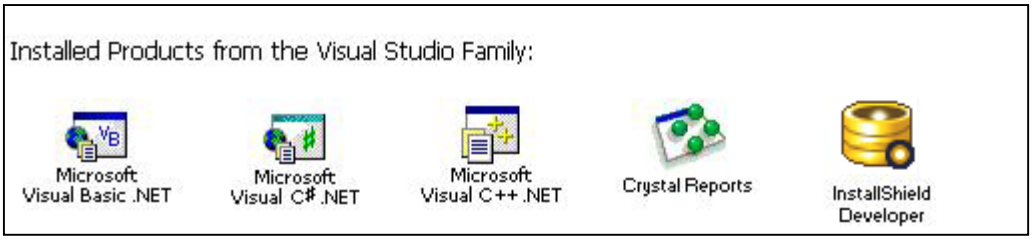
Working with Visual Studio .NET and InstallShield Developer

Using InstallShield Developer's Integration with Visual Studio .NET

InstallShield Developer 8 introduces powerful new integration with the Visual Studio .NET IDE that greatly increases ease of use and productivity. InstallShield Developer's Visual Studio integration makes your InstallShield project part of your Visual Studio .NET development solution. This allows you to:

- Build your setup individually, or as part of the rest of your development project.
- Edit all aspects of your setup project directly in the Visual Studio IDE along with the rest of your solution.
- Automatically include new elements of your solution to your setup project.
- Dynamically acquire any dependencies every time the solution is built.

To utilize InstallShield Developer's Visual Studio .NET integration, you must have Visual Studio .NET installed before you install InstallShield Developer. InstallShield Developer will automatically detect that Visual Studio .NET is installed on the machine and install the required elements for integration with the Visual Studio IDE. The next time you start Visual Studio .NET, the InstallShield logo will be displayed as part of the startup screen.



ABOVE: The Visual Studio .NET startup screen after InstallShield Developer is installed.

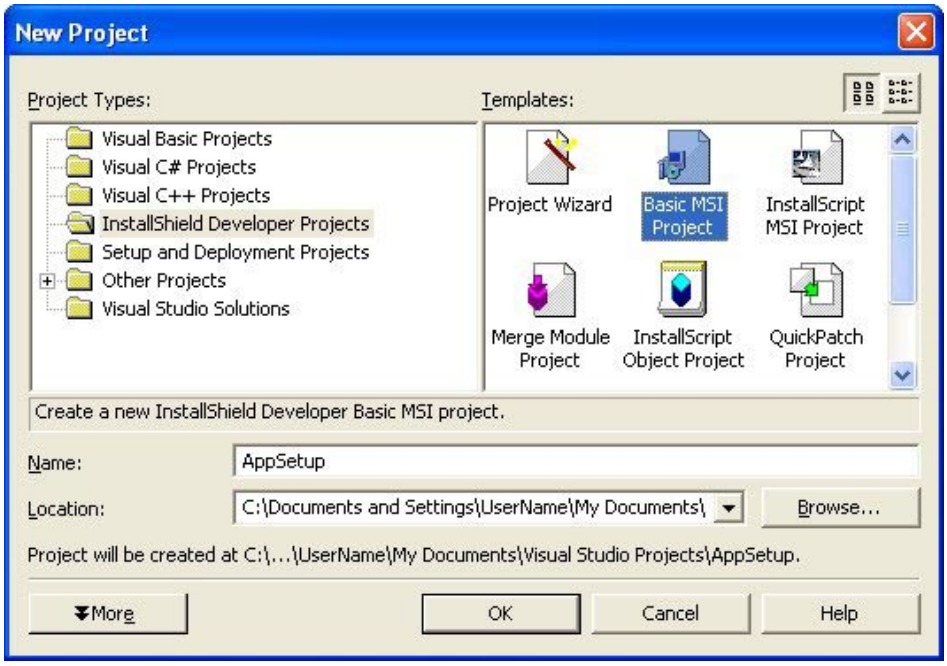
Adding an InstallShield Project to your Visual Studio .NET Solution

The first step in working with InstallShield Developer's Visual Studio integration is to add an InstallShield project to your current solution.

1. Right-click on the root of your solution tree and select **Add ▸ New Project**

2. Select **InstallShield Developer Projects** from the **Add New Project** window.

3. Choose the type of InstallShield project you wish to add from the templates listed on the right side of the window.



ABOVE: The New Project window

The easiest way to add an InstallShield project to your solution is to select the **Project Wizard** project type from the **Add New Project** window. This will launch the InstallShield Visual Studio .NET Project Wizard, which will guide you through the process of creating a basic InstallShield project and linking your solution's output groups to the project.

1. **Select whether you wish to use a Standard Project or Basic MSI Project**.
   In most cases, you will use a Basic MSI Project unless your setup will require InstallScript functionality.

2. **Provide the application's name, version, and default installation directory.**
   You can change all of this later inside of the IDE.

3. **Select whether or not the application will utilize the InstallShield Update Service.**
   Further information about the service is available at http://www.installshield.com/isus/.

4. **Enter your company information**.
   This will be displayed in the Windows Add/Remove Programs control panel.

5. **Select languages.**
   Choose which languages, if any, your setup will support besides the primary language you selected during the Developer installation process. Language Packs (sold separately) are required to support additional languages in the same installation.

6. **Create your feature choices and organization**.
   If you're not sure which features you will need, just keep the defaults and you can change the feature layout later in the IDE.

7. **Associate your project outputs with features.**
   All of the default project outputs will be displayed, even if you're not using them. At a minimum, you will want to associate your primary output with one feature. (More information on associating project outputs is included in the section All About Project Outputs.)

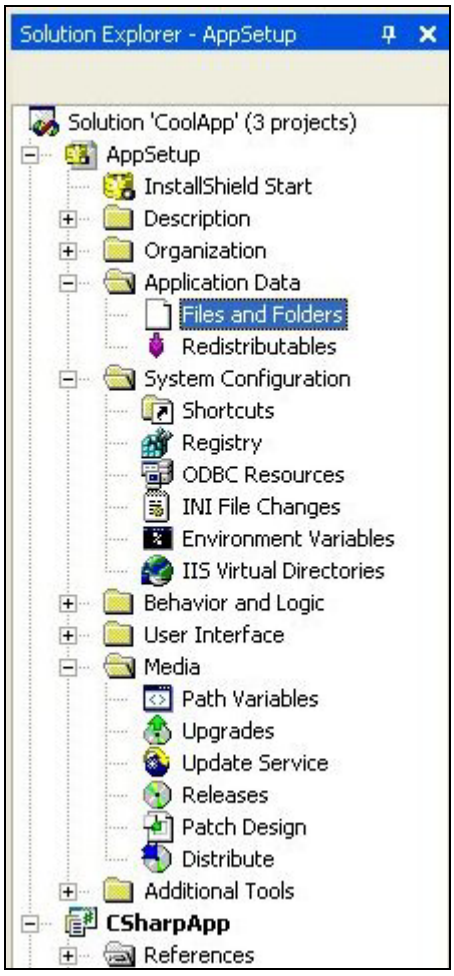
8. **Associate any additional files with features.**
   If you have files that are not part of your solution (e.g., Readme.txt), you can select those files here and associate them with features.

9. **Create Shortcuts.**
   If you select your project's primary output as the target of the shortcut, an Advertisable Shortcut will automatically be created for the project's executable.

10. **Import any additional registry data.**
    You don't need to import any COM or .NET registry data; this information will be automatically included in your setup as part of the build process.

11. **Select default dialogs.**
    You will be able to edit these dialogs and/or include custom dialogs inside of the IDE.

    Once you complete the Project Wizard, your InstallShield project will be listed as part of your solution in the Solution Explorer. Each node on the tree under the InstallShield project represents a different view from the InstallShield IDE. These views work just like they would in the InstallShield IDE, so there is no need to learn a different interface.



ABOVE: The Solution Explorer with an InstallShield project (called AppSetup) added to the solution.

All About Project Outputs

Since the files that make up your solution are likely to change frequently, InstallShield uses the solution's project outputs as place-holders for the files that your solution will generate and ultimately be included in the setup. When you associate a project's output with an InstallShield project, InstallShield will automatically include all the files as part of that output group as well as dynamically acquire any dependencies, every time you build your solution.

Use the **Application Data ▸ Files** view to associate project outputs with your InstallShield project.

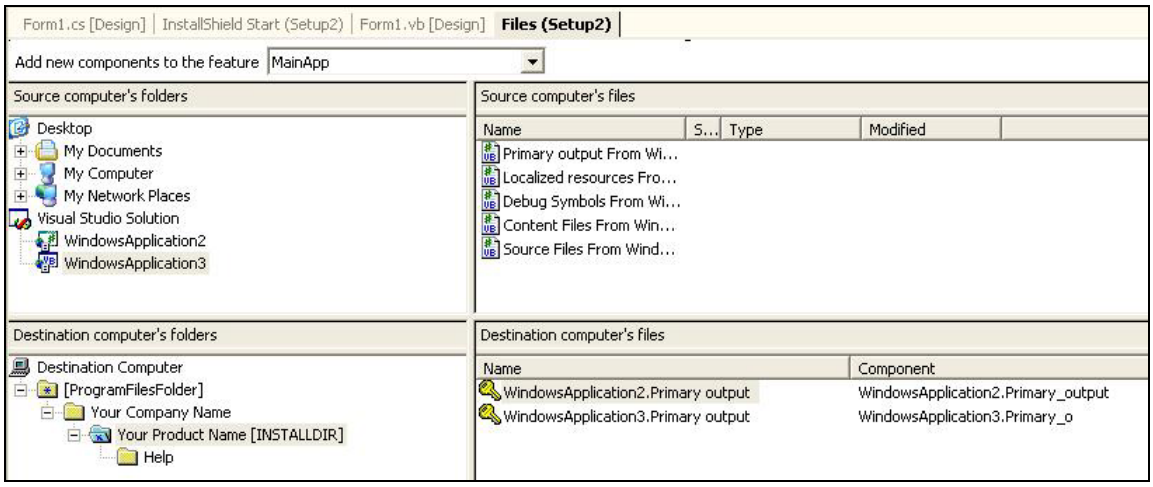
1. Use the **Destination computer's folders** frame to create the necessary destination folders on the target machine.

2. Select the project you want to work with under the **Visual Studio Solution** node of the **Source computer's folder** frame.

3. Select which feature you want to associate the project output with at the very top of the frame.

4. Select the output group you want to associate from the **Source computer's files** frame and drag-and-drop to the appropriate folder on the **Destination computer's folders** frame.

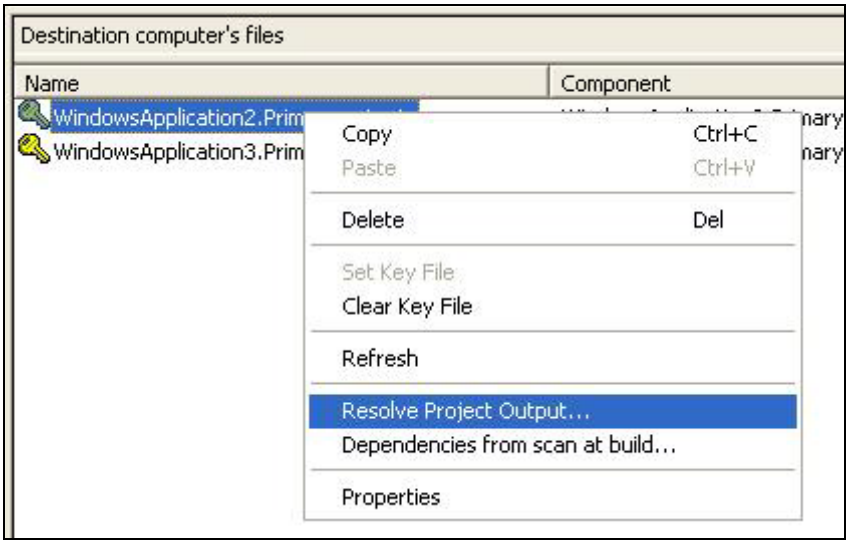
5. Continue this process for all projects and output groups you want to include in your InstallShield setup project.



ABOVE: The files view inside of the Visual Studio .NET IDE.

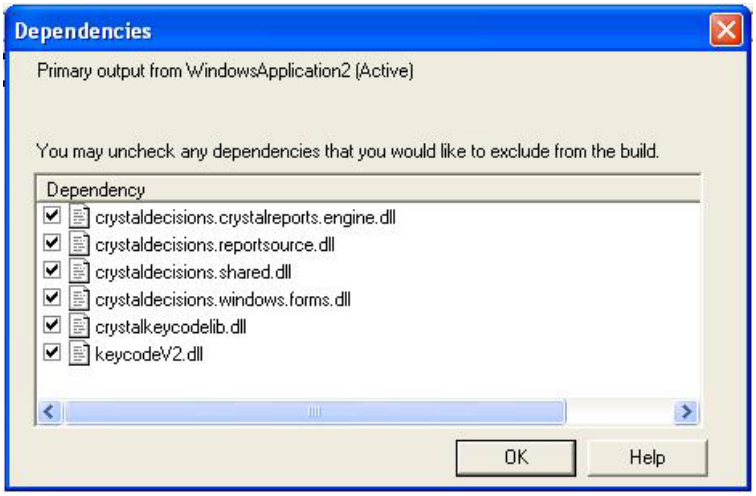
It is important to note that InstallShield will list all the default output groups for a particular project even if you're not using that output group. For example, InstallShield will list the Localized output group as an option available for association with your project, even if you are not creating any localized versions of your application. It is up to you to ensure that you include all the necessary output groups as part of your InstallShield project.

If you want to review the files that are part of an output group, or view which dependencies are being brought into the setup by a particular output group, you can do so by right-clicking on the output group in the **Destination computer's files** pane and selecting the appropriate menu item.



ABOVE: The right-click menu on a Visual Studio output group.

InstallShield will automatically scan an included output group at build-time and include any necessary dependencies. The Merge Module Search Path will be searched for the dependencies, and if an appropriate Merge Module is found, it will be included in the project. Otherwise, the individual files that your project depends on will be included in the setup. If you want to view the files that your project is dependent on, select the **Dependencies from scan at build** and InstallShield will run a dependency scan and display the results in the Dependencies window. You can also override dependency inclusion in the Dependencies window by un-checking dependent files you don't want to include in the build.

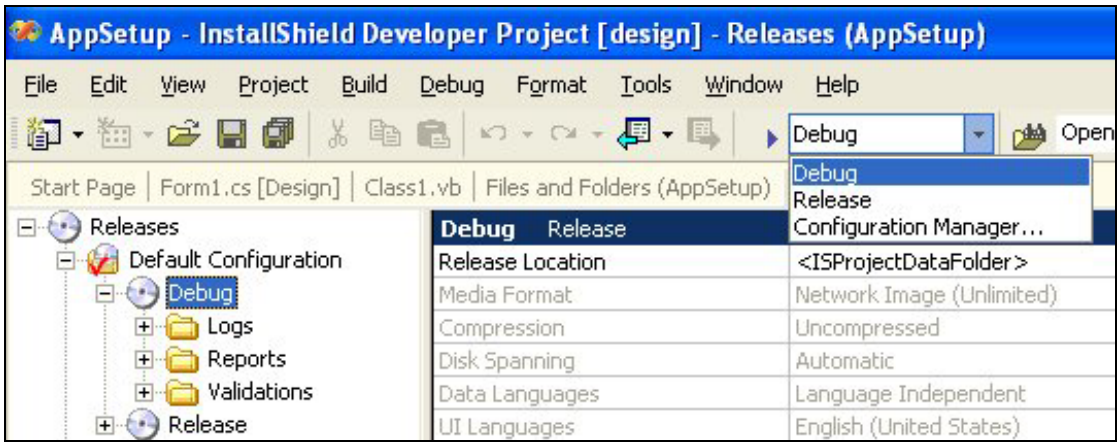


ABOVE: The dependencies list of a project that is dependent on Crystal Reports.

Building Your InstallShield Developer Project

Your InstallShield Project can be built separately by right-clicking on the Setup node in the Solution Explorer and by selecting the appropriate build option. Your setup will also be re-built every time you initiate a re-build of your entire solution. In either case, the build log will be sent to the Output window in the Visual Studio IDE, so you can see the results of your build and any errors or warnings that have been generated.

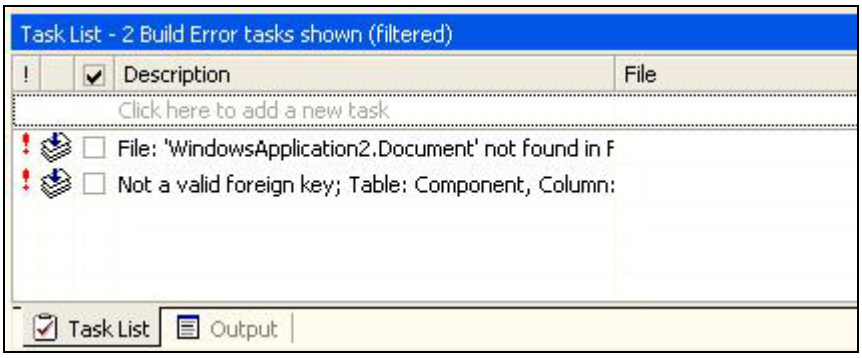
The release-types available as part of your InstallShield project automatically include any configurations that you have specified in the Visual Studio Configuration Manager. By default, there are two configurations: Debug and Release. So, if you select to build the Debug release type, InstallShield will automatically include all the project configurations that you have specified in the Configuration Manager for the Debug configuration.



ABOVE: The Releases View in InstallShield with the Debug type selected.

The Task List

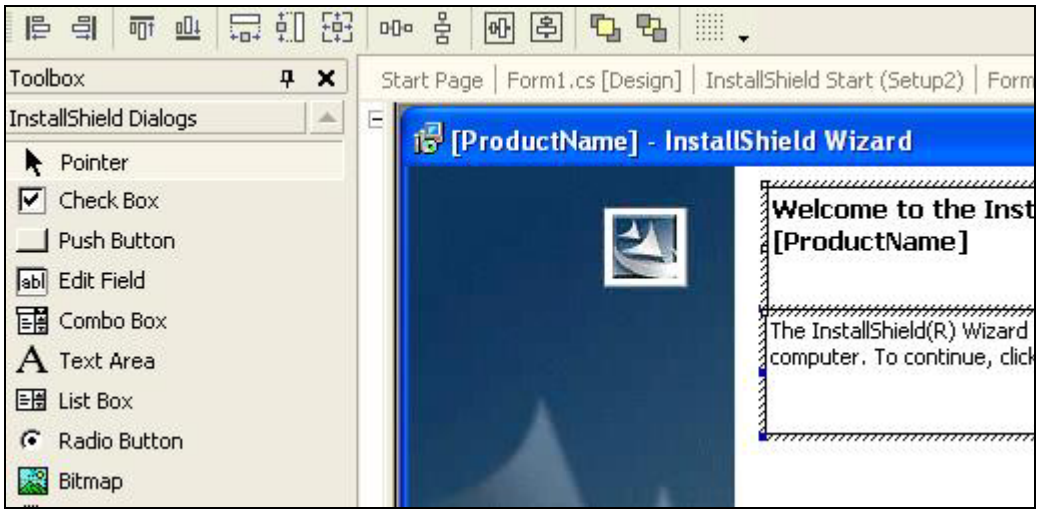
In addition to utilizing the Visual Studio Output Window, InstallShield also makes full use of the Task Window. Whenever your build or validation produces any warnings or errors, an item will be created on the task list for you to follow-up on later. In many cases, the item will have a hyperlink to more information about that error/warning, should you require additional assistance.



ABOVE: A view of the task list after validating a setup.

Editing InstallShield Dialogs in the Visual Studio IDE

Since the Windows Installer utilizes a special dialog format and has many controls that are unique to the Windows Installer UI, InstallShield provides its own dialog editor separate from the Visual Studio dialog editor. Whenever you select to edit a dialog, the Visual Studio Toolbox automatically loads with the InstallShield Dialogs tab and the InstallShield Layout toolbar is also activated, giving you all the tools you need to edit a Windows Installer dialog.



ABOVE: A view of a dialog (right) with the InstallShield Dialogs toolbox (left) and the InstallShield Layout toolbar (top).

In addition to the InstallShield Layout toolbar, you can activate and de-activate other InstallShield toolbars by right-clicking in the toolbar area and selecting them from the menu:

- **InstallScript** — Allows you to navigate through the various InstallScript files that are part of your project.
- **InstallShield** — Provides access to all basic InstallShield functionality, such as running the various wizards and accessing InstallShield's default options.
- **InstallShield Layout** — Allows you to change the layout of elements on a dialog.
- **MSI Debugger** — Gives you the ability to add, remove, and monitor breakpoints in your project.

In most cases, the appropriate toolbar will be displayed and activated automatically for you, so you won't need to turn these on and off manually.

Advanced Options

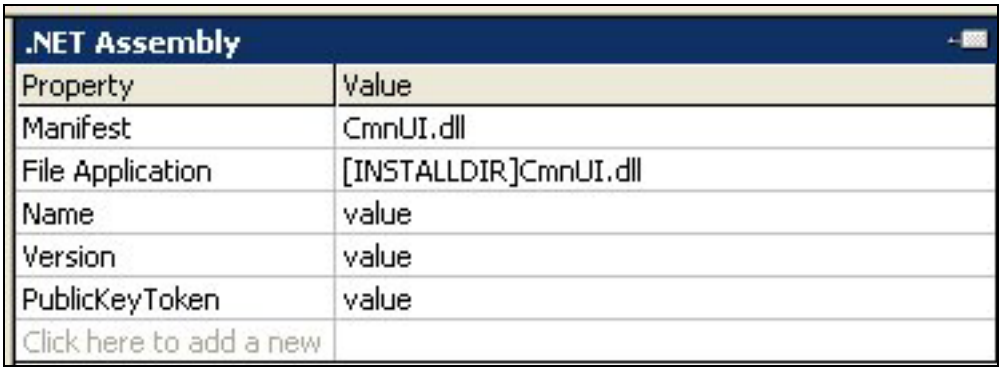
Creating an Installation for a .NET Assembly (Manual Method)

1. Create a new project or open your currently existing project in InstallShield Developer.
2. In the Setup Design view, create a component to hold your .NET Assembly.
3. Add the file(s) belonging to the .NET Assembly to the component's file list.
4. Right-click on the Assembly node in the Advanced Settings of your component and select **New .NET Assembly**.
5. Select the .NET Assembly node and examine the properties. You'll see that your .DLL or .EXE has automatically been selected as the Manifest file.
6. Select whether you want the Assembly to be installed to the Global Assembly Cache or not by setting the **File Application** property.

   Tip: Your Assembly must be properly signed in order to be correctly installed into the Global Assembly Cache. See the .NET documentation for more information.
7. Set the Name, Version, and PublicKeyToken properties. If your Assembly contains other properties, you may add them by clicking on the last row of the property list for your Assembly.

   Tip: If your Assembly does not have PublicKeyToken, you'll want to go into the Direct Editor and remove the PublicKeyToken value from the MsiAssemblyName table for your Assembly.

   Tip: The InstallShield Developer online help provides sample code to assist you in extracting the correct property names and values from your .NET Assembly.



Above: A look at the .NET Assembly properties in InstallShield Developer.

Congratulations! You have created a component for a .NET Assembly. Repeat the above process for each .NET Assembly in your project.

Creating an Installation for a Side-by-Side (Win32) Assembly

A Side-by-Side (Win32) Assembly usually consists of a single Side-by-Side Portable Executable (.EXE, .DLL, .OCX, etc …) and its accompanying .MANIFEST file. As mentioned previously, Side-by-Side components only operate Side-by-Side on Windows XP, so you must take special care to author your installation to work on down-level platforms if you intend to support them. Side-by-Side components can be configured for isolated application installation or to the Global Assembly Cache.

Below are general steps necessary for creating a Side-by-Side Assembly component:

1. Create a new project or open your currently existing project in InstallShield Developer.

   Tip: Be sure to create an Install Condition in your Product Properties if your Side-by-Side components do not support down-level platforms.

2. In the Setup Design view, create a component to hold your Side-by-Side assembly.

3. Add the file(s) belonging to the Assembly to the component's file list.

   Tip: If you do not set a key file, InstallShield will automatically set the .MANIFEST file as the key file in the next step.

4. Right-click on the Assembly node in the Advanced Settings of your component and select **New Win32 Assembly**.

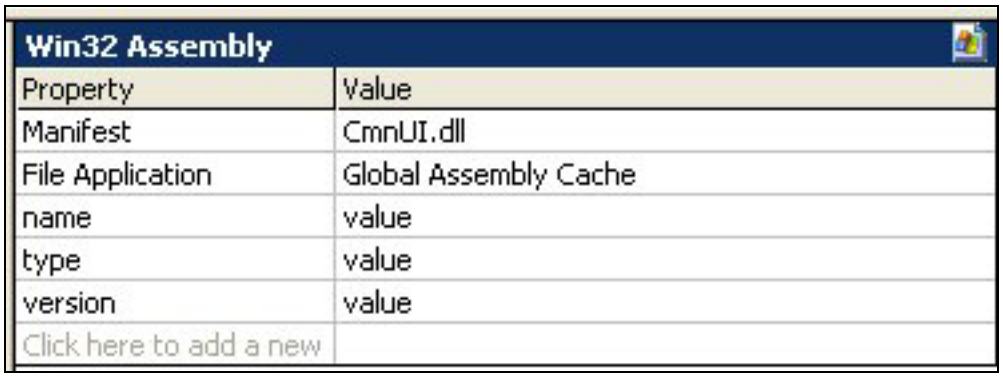
5. Select the Win32 Assembly node and examine the properties.
   You'll see that your .MANIFEST file has automatically been selected as the Manifest file.

   Tip: If you plan to target down-level platforms, be sure to include all the COM data that is part of your assembly in the COM Registration section of the component's Advanced Settings view. You can also use InstallShield's **extract COM information at build** or **component wizard** functionality to automatically extract the required COM data for down-level platforms. (See InstallShield Developer's online help for more information on this functionality.) Note that on Windows XP this information will be ignored and the manifest data will be used instead.

6. Select whether you want the Assembly to be installed to the Global Assembly Cache or not by setting the **Global Cache** property.

   Tip: Your Assembly must be properly signed in order to be correctly installed into the Global Assembly Cache. See the Windows Platform SDK for more information.

7. Set the Name, Type, and Version properties. These values must be copied exactly as they appear in the assembly manifest. If your assembly contains other properties, you may add them by clicking on the last row of the property list for your assembly.



Win32 Assembly	
Property	Value
Manifest	CmnUI.dll
File Application	Global Assembly Cache
name	value
type	value
version	value
Click here to add a new	

Above: A look at the Win32 Assembly properties in InstallShield Developer.

Congratulations! You have created a component for a Side-by-Side Assembly. Repeat the above process for each Side-by-Side Assembly in your project.

About InstallShield's .NET Solutions

InstallShield Developer is the most comprehensive .NET solution available for managed, non-managed, and inter-operable code deployment. For more information about InstallShield's .NET installation solutions, please visit http://www.installshield.com/products/dotnet.asp.