**Assemblies in the Common Language Runtime**

This page is specific to

**Microsoft Visual Studio 2008/.NET Framework 3.5**

Other versions are also available for the following:

[Microsoft Visual Studio 2003/.NET Framework 1.1](http://msdn.microsoft.com/en-us/library/hk5f40ct(VS.71).aspx)

[Microsoft Visual Studio 2005/.NET Framework 2.0](http://msdn.microsoft.com/en-us/library/hk5f40ct(VS.80).aspx)

[.NET Framework 3.0](http://msdn.microsoft.com/en-us/library/hk5f40ct(VS.85).aspx)

Assemblies are the building blocks of .NET Framework applications; they form the fundamental unit of deployment, version control, reuse, activation scoping, and security permissions. An assembly is a collection of types and resources that are built to work together and form a logical unit of functionality. An assembly provides the common language runtime with the information it needs to be aware of type implementations. To the runtime, a type does not exist outside the context of an assembly.

In This Section

[Assemblies Overview](http://msdn.microsoft.com/en-us/library/k3677y81.aspx)

Provides an overview of the functions performed by assemblies.

[Assembly Benefits](http://msdn.microsoft.com/en-us/library/6h38y9z9.aspx)

Describes how assemblies help solve versioning problems and DLL conflicts.

[Assembly Contents](http://msdn.microsoft.com/en-us/library/zst29sk2.aspx)

Describes the elements that make up an assembly.

[Assembly Manifest](http://msdn.microsoft.com/en-us/library/1w45z383.aspx)

Describes the data in the assembly manifest, and how it is stored in assemblies.

[Global Assembly Cache](http://msdn.microsoft.com/en-us/library/yf1d93sz.aspx)

Describes the global assembly cache and how it is used with assemblies.

[Strong-Named Assemblies](http://msdn.microsoft.com/en-us/library/wd40t7ad.aspx)

Describes the characteristics of strong-named assemblies.

[Assembly Security Considerations](http://msdn.microsoft.com/en-us/library/ab4eace3.aspx)

Discusses how security works with assemblies.

[Assembly Versioning](http://msdn.microsoft.com/en-us/library/51ket42z.aspx)

Provides an overview of the .NET Framework versioning policy.

[Assembly Placement](http://msdn.microsoft.com/en-us/library/2h3sywsc.aspx)

Discusses where to locate assemblies.

[Assemblies and Side-by-Side Execution](http://msdn.microsoft.com/en-us/library/fdhkd3a5.aspx)

Provides an overview of using multiple versions of the runtime or of an assembly simultaneously.

**Assemblies Overview**

Assemblies are a fundamental part of programming with the .NET Framework. An assembly performs the following functions:

* It contains code that the common language runtime executes. Microsoft intermediate language (MSIL) code in a portable executable (PE) file will not be executed if it does not have an associated assembly manifest. Note that each assembly can have only one entry point (that is, **DllMain**, **WinMain**, or **Main**).
* It forms a security boundary. An assembly is the unit at which permissions are requested and granted. For more information about security boundaries as they apply to assemblies, see [Assembly Security Considerations](http://msdn.microsoft.com/en-us/library/ab4eace3.aspx).
* It forms a type boundary. Every type's identity includes the name of the assembly in which it resides. A type called MyType loaded in the scope of one assembly is not the same as a type called MyType loaded in the scope of another assembly.
* It forms a reference scope boundary. The assembly's manifest contains assembly metadata that is used for resolving types and satisfying resource requests. It specifies the types and resources that are exposed outside the assembly. The manifest also enumerates other assemblies on which it depends.
* It forms a version boundary. The assembly is the smallest versionable unit in the common language runtime; all types and resources in the same assembly are versioned as a unit. The assembly's manifest describes the version dependencies you specify for any dependent assemblies. For more information about versioning, see [Assembly Versioning](http://msdn.microsoft.com/en-us/library/51ket42z.aspx).
* It forms a deployment unit. When an application starts, only the assemblies that the application initially calls must be present. Other assemblies, such as localization resources or assemblies containing utility classes, can be retrieved on demand. This allows applications to be kept simple and thin when first downloaded. For more information about deploying assemblies, see [Deploying Applications](http://msdn.microsoft.com/en-us/library/6hbb4k3e.aspx).
* It is the unit at which side-by-side execution is supported. For more information about running multiple versions of an assembly, see [Assemblies and Side-by-Side Execution](http://msdn.microsoft.com/en-us/library/fdhkd3a5.aspx).

Assemblies can be static or dynamic. Static assemblies can include .NET Framework types (interfaces and classes), as well as resources for the assembly (bitmaps, JPEG files, resource files, and so on). Static assemblies are stored on disk in portable executable (PE) files. You can also use the .NET Framework to create dynamic assemblies, which are run directly from memory and are not saved to disk before execution. You can save dynamic assemblies to disk after they have executed.

There are several ways to create assemblies. You can use development tools, such as Visual Studio 2005, that you have used in the past to create .dll or .exe files. You can use tools provided in the Windows Software Development Kit (SDK) to create assemblies with modules created in other development environments. You can also use common language runtime APIs, such as [Reflection.Emit](http://msdn.microsoft.com/en-us/library/system.reflection.emit.aspx), to create dynamic assemblies.

**Assembly Security Considerations**

* When you build an assembly, you can specify a set of permissions that the assembly requires to run. Whether certain permissions are granted or not granted to an assembly is based on evidence.

There are two distinct ways evidence is used:

* The input evidence is merged with the evidence gathered by the loader to create a final set of evidence used for policy resolution. The methods that use this semantic include **Assembly.Load**, **Assembly.LoadFrom**, and **Activator.CreateInstance**.
* The input evidence is used unaltered as the final set of evidence used for policy resolution. The methods that use this semantic include **Assembly.Load(byte[])** and **AppDomain.DefineDynamicAssembly()**.

Optional permissions can be granted by the [security policy](http://msdn.microsoft.com/en-us/library/33tceax8.aspx) set on the computer where the assembly will run. If you want your code to handle all potential security exceptions, you can do one of the following:

* Insert a permission request for all the permissions your code must have, and handle up front the load-time failure that occurs if the permissions are not granted.
* Do not use a permission request to obtain permissions your code might need, but be prepared to handle security exceptions if permissions are not granted.

|  |
| --- |
| **NoteNote:** |
| Security is a complex area, and you have many options to choose from. For more information, see [Key Security Concepts](http://msdn.microsoft.com/en-us/library/z164t8hs.aspx). |

At load time, the assembly's evidence is used as input to security policy. Security policy is established by the enterprise and the computer's administrator as well as by user policy settings, and determines the set of permissions that is granted to all managed code when executed. Security policy can be established for the publisher of the assembly (if it has a signing tool generated signature), for the Web site and zone (in Internet Explorer terms) the assembly was downloaded from, or for the assembly's strong name. For example, a computer's administrator can establish security policy that allows all code downloaded from a Web site and signed by a given software company to access a database on a computer, but does not grant access to write to the computer's disk.

http://i.msdn.microsoft.com/Global/Images/clear.gif Strong-Named Assemblies and Signing Tools

You can sign an assembly in two different but complementary ways: with a strong name or using either the [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) in .NET Framework version 1.0 and 1.1 or [Sign Tool (SignTool.exe)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx) in later versions of the .NET Framework. Signing an assembly with a strong name adds a public key encryption to the file containing the assembly manifest. Strong name signing helps to verify name uniqueness, prevent name spoofing, and provide callers with some identity when a reference is resolved.

However, no level of trust is associated with a strong name, which makes the [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) and [Sign Tool (SignTool.exe)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx) important. The two signing tools require a publisher to prove its identity to a third-party authority and obtain a certificate. This certificate is then embedded in your file and can be used by an administrator to decide whether to trust the code's authenticity.

You can give both a strong name and a digital signature created using the [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) or [Sign Tool (SignTool.exe)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx) to an assembly, or you can use either alone. The two signing tools can sign only one file at a time; for a multifile assembly, you sign the file that contains the assembly manifest. A strong name is stored in the file containing the assembly manifest, but a signature created using the [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) or [Sign Tool (SignTool.exe)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx) is stored in a reserved slot in the portable executable (PE) file containing the assembly manifest. Signing of an assembly using the [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) or [Sign Tool (SignTool.exe)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx) can be used (with or without a strong name) when you already have a trust hierarchy that relies on [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) or [Sign Tool (SignTool.exe)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx) generated signatures, or when your policy uses only the key portion and does not check a chain of trust.

|  |
| --- |
| **NoteNote:** |
| When using both a strong name and a signing tool signature on an assembly, the strong name must be assigned first. |

The common language runtime also performs a hash verification; the assembly manifest contains a list of all files that make up the assembly, including a hash of each file as it existed when the manifest was built. As each file is loaded, its contents are hashed and compared with the hash value stored in the manifest. If the two hashes do not match, the assembly fails to load.

Because strong naming and signing using the [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) or [Sign Tool (SignTool.exe)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx) guarantee integrity, you can base code access security policy on these two forms of assembly evidence. Strong naming and signing using the [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) or [Sign Tool (SignTool.exe)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx) guarantee integrity through digital signatures and certificates. All the technologies mentioned—hash verification, strong naming, and signing using the [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) or [Sign Tool (SignTool.exe)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx)—work together to ensure that the assembly has not been altered in any way.

**Assembly Benefits**

Assemblies are designed to simplify application deployment and to solve versioning problems that can occur with component-based applications.

End users and developers are familiar with versioning and deployment issues that arise from today's component-based systems. Some end users have experienced the frustration of installing a new application on their computer, only to find that an existing application has suddenly stopped working. Many developers have spent countless hours trying to keep all necessary registry entries consistent in order to activate a COM class.

Many deployment problems have been solved by the use of assemblies in the .NET Framework. Because they are self-describing components that have no dependencies on registry entries, assemblies enable zero-impact application installation. They also simplify uninstalling and replicating applications.

http://i.msdn.microsoft.com/Global/Images/clear.gif Versioning Problems

Currently two versioning problems occur with Win32 applications:

* Versioning rules cannot be expressed between pieces of an application and enforced by the operating system. The current approach relies on backward compatibility, which is often difficult to guarantee. Interface definitions must be static, once published, and a single piece of code must maintain backward compatibility with previous versions. Furthermore, code is typically designed so that only a single version of it can be present and executing on a computer at any given time.
* There is no way to maintain consistency between sets of components that are built together and the set that is present at run time.

These two versioning problems combine to create DLL conflicts, where installing one application can inadvertently break an existing application because a certain software component or DLL was installed that was not fully backward compatible with a previous version. Once this situation occurs, there is no support in the system for diagnosing and fixing the problem.

http://i.msdn.microsoft.com/Global/Images/clear.gif An End to DLL Conflicts

Microsoft® Windows® 2000 began to fully address these problems. It provides two features that partially fix DLL conflicts:

* Windows 2000 enables you to create client applications where the dependent .dll files are located in the same directory as the application's .exe file. Windows 2000 can be configured to check for a component in the directory where the .exe file is located before checking the fully qualified path or searching the normal path. This enables components to be independent of components installed and used by other applications.
* Windows 2000 locks files that are shipped with the operating system in the System32 directory so they cannot be inadvertently replaced when applications are installed.

The common language runtime uses assemblies to continue this evolution toward a complete solution to DLL conflicts.

http://i.msdn.microsoft.com/Global/Images/clear.gif The Assembly Solution

To solve versioning problems, as well as the remaining problems that lead to DLL conflicts, the runtime uses assemblies to do the following:

* Enable developers to specify version rules between different software components.
* Provide the infrastructure to enforce versioning rules.
* Provide the infrastructure to allow multiple versions of a component to be run simultaneously (called side-by-side execution).

**Assembly Contents**

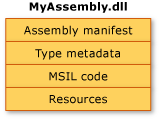
In general, a static assembly can consist of four elements:

* The [assembly manifest](http://msdn.microsoft.com/en-us/library/1w45z383.aspx), which contains assembly metadata.
* Type metadata.
* Microsoft intermediate language (MSIL) code that implements the types.
* A set of resources.

Only the assembly manifest is required, but either types or resources are needed to give the assembly any meaningful functionality.

There are several ways to group these elements in an assembly. You can group all elements in a single physical file, which is shown in the following illustration.

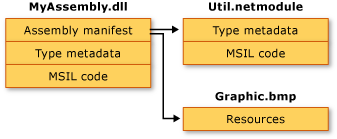
**Single-file assembly**



Alternatively, the elements of an assembly can be contained in several files. These files can be modules of compiled code (.netmodule), resources (such as .bmp or .jpg files), or other files required by the application. Create a multifile assembly when you want to combine modules written in different languages and to optimize downloading an application by putting seldom used types in a module that is downloaded only when needed.

In the following illustration, the developer of a hypothetical application has chosen to separate some utility code into a different module and to keep a large resource file (in this case a .bmp image) in its original file. The .NET Framework downloads a file only when it is referenced; keeping infrequently referenced code in a separate file from the application optimizes code download.

**Multifile assembly**



|  |
| --- |
| **NoteNote:** |
| The files that make up a multifile assembly are not physically linked by the file system. Rather, they are linked through the assembly manifest and the common language runtime manages them as a unit. |

In this illustration, all three files belong to an assembly, as described in the assembly manifest contained in MyAssembly.dll. To the file system, they are three separate files. Note that the file Util.netmodule was compiled as a module because it contains no assembly information. When the assembly was created, the assembly manifest was added to MyAssembly.dll, indicating its relationship with Util.netmodule and Graphic.bmp.

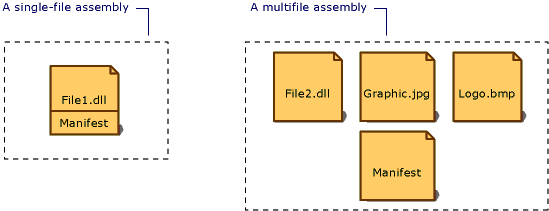
As you currently design your source code, you make explicit decisions about how to partition the functionality of your application into one or more files. When designing .NET Framework code, you will make similar decisions about how to partition the functionality into one or more assemblies.

**Assembly Manifest**

Every assembly, whether static or dynamic, contains a collection of data that describes how the elements in the assembly relate to each other. The assembly manifest contains this assembly metadata. An assembly manifest contains all the metadata needed to specify the assembly's version requirements and security identity, and all metadata needed to define the scope of the assembly and resolve references to resources and classes. The assembly manifest can be stored in either a PE file (an .exe or .dll) with Microsoft intermediate language (MSIL) code or in a standalone PE file that contains only assembly manifest information.

The following illustration shows the different ways the manifest can be stored.

**Types of assemblies**



For an assembly with one associated file, the manifest is incorporated into the PE file to form a single-file assembly. You can create a multifile assembly with a standalone manifest file or with the manifest incorporated into one of the PE files in the assembly.

Each assembly's manifest performs the following functions:

* Enumerates the files that make up the assembly.
* Governs how references to the assembly's types and resources map to the files that contain their declarations and implementations.
* Enumerates other assemblies on which the assembly depends.
* Provides a level of indirection between consumers of the assembly and the assembly's implementation details.
* Renders the assembly self-describing.

http://i.msdn.microsoft.com/Global/Images/clear.gif Assembly Manifest Contents

The following table shows the information contained in the assembly manifest. The first four items—the assembly name, version number, culture, and strong name information—make up the assembly's identity.

|  |  |
| --- | --- |
| **Information** | **Description** |
| Assembly name | A text string specifying the assembly's name. |
| Version number | A major and minor version number, and a revision and build number. The common language runtime uses these numbers to enforce version policy. |
| Culture | Information on the culture or language the assembly supports. This information should be used only to designate an assembly as a satellite assembly containing culture- or language-specific information. (An assembly with culture information is automatically assumed to be a satellite assembly.) |
| Strong name information | The public key from the publisher if the assembly has been given a strong name. |
| List of all files in the assembly | A hash of each file contained in the assembly and a file name. Note that all files that make up the assembly must be in the same directory as the file containing the assembly manifest. |
| Type reference information | Information used by the runtime to map a type reference to the file that contains its declaration and implementation. This is used for types that are exported from the assembly. |
| Information on referenced assemblies | A list of other assemblies that are statically referenced by the assembly. Each reference includes the dependent assembly's name, assembly metadata (version, culture, operating system, and so on), and public key, if the assembly is strong named. |

You can add or change some information in the assembly manifest by using assembly attributes in your code. You can change version information and informational attributes, including Trademark, Copyright, Product, Company, and Informational Version. For a complete list of assembly attributes, see [Setting Assembly Attributes](http://msdn.microsoft.com/en-us/library/4w8c1y2s.aspx).

**Global Assembly Cache**

Each computer where the common language runtime is installed has a machine-wide code cache called the global assembly cache. The global assembly cache stores assemblies specifically designated to be shared by several applications on the computer.

You should share assemblies by installing them into the global assembly cache only when you need to. As a general guideline, keep assembly dependencies private, and locate assemblies in the application directory unless sharing an assembly is explicitly required. In addition, it is not necessary to install assemblies into the global assembly cache to make them accessible to COM interop or unmanaged code.

|  |
| --- |
| **NoteNote:** |
| There are scenarios where you explicitly do not want to install an assembly into the global assembly cache. If you place one of the assemblies that make up an application in the global assembly cache, you can no longer replicate or install the application by using the **xcopy** command to copy the application directory. You must move the assembly in the global assembly cache as well. |

There are several ways to deploy an assembly into the global assembly cache:

* Use an installer designed to work with the global assembly cache. This is the preferred option for installing assemblies into the global assembly cache.
* Use a developer tool called the [Global Assembly Cache tool (Gacutil.exe)](http://msdn.microsoft.com/en-us/library/ex0ss12c.aspx), provided by the Windows Software Development Kit (SDK).
* Use Windows Explorer to drag assemblies into the cache.

|  |
| --- |
| **NoteNote:** |
| In deployment scenarios, use Windows Installer 2.0 to install assemblies into the global assembly cache. Use Windows Explorer or the Global Assembly Cache tool only in development scenarios, because they do not provide assembly reference counting and other features provided when using the Windows Installer. |

Administrators often protect the systemroot directory using an access control list (ACL) to control write and execute access. Because the global assembly cache is installed in a subdirectory of the systemroot directory, it inherits that directory's ACL. It is recommended that only users with Administrator privileges be allowed to delete files from the global assembly cache.

Assemblies deployed in the global assembly cache must have a strong name. When an assembly is added to the global assembly cache, integrity checks are performed on all files that make up the assembly. The cache performs these integrity checks to ensure that an assembly has not been tampered with, for example, when a file has changed but the manifest does not reflect the change.

**Strong-Named Assemblies**

A strong name consists of the assembly's identity—its simple text name, version number, and culture information (if provided)—plus a public key and a digital signature. It is generated from an assembly file (the file that contains the assembly manifest, which in turn contains the names and hashes of all the files that make up the assembly), using the corresponding private key. Microsoft® Visual Studio® .NET and other development tools provided in the Windows Software Development Kit (SDK) can assign strong names to an assembly. Assemblies with the same strong name are expected to be identical.

You can ensure that a name is globally unique by signing an assembly with a strong name. In particular, strong names satisfy the following requirements:

* Strong names guarantee name uniqueness by relying on unique key pairs. No one can generate the same assembly name that you can, because an assembly generated with one private key has a different name than an assembly generated with another private key.
* Strong names protect the version lineage of an assembly. A strong name can ensure that no one can produce a subsequent version of your assembly. Users can be sure that a version of the assembly they are loading comes from the same publisher that created the version the application was built with.
* Strong names provide a strong integrity check. Passing the .NET Framework security checks guarantees that the contents of the assembly have not been changed since it was built. Note, however, that strong names in and of themselves do not imply a level of trust like that provided, for example, by a digital signature and supporting certificate.

When you reference a strong-named assembly, you expect to get certain benefits, such as versioning and naming protection. If the strong-named assembly then references an assembly with a simple name, which does not have these benefits, you lose the benefits you would derive from using a strong-named assembly and revert to DLL conflicts. Therefore, strong-named assemblies can only reference other strong-named assemblies.

**How to: Sign an Assembly with a Strong Name**

The Windows Software Development Kit (SDK) provides several ways to sign an assembly with a strong name:

* Using the [Assembly Linker (Al.exe)](http://msdn.microsoft.com/en-us/library/c405shex.aspx) provided by the Windows SDK.
* Using assembly attributes to insert the strong name information in your code. You can use either the [AssemblyKeyFileAttribute](http://msdn.microsoft.com/en-us/library/system.reflection.assemblykeyfileattribute.aspx) or the [AssemblyKeyNameAttribute](http://msdn.microsoft.com/en-us/library/system.reflection.assemblykeynameattribute.aspx), depending on where the key file to be used is located.

|  |
| --- |
| **NoteNote:** |
| In the .NET Framework version 2.0, some compilers issue warning messages when attributes are used. |

* Using compiler options such **/keyfile** or **/delaysign** in C# and Visual Basic, or the **/KEYFILE** or **/DELAYSIGN** linker option in C++. (For information on delay signing, see [Delay Signing an Assembly](http://msdn.microsoft.com/en-us/library/t07a3dye.aspx).)

|  |
| --- |
| **NoteNote:** |
| In Visual Studio 2005, the development environment provides tools for signing assemblies. See [Managing Assembly and Manifest Signing](http://msdn.microsoft.com/en-us/library/ms247066.aspx) and [Signing Page, Project Designer](http://msdn.microsoft.com/en-us/library/0k50fs3b.aspx). |

You must have a cryptographic key pair to sign an assembly with a strong name. For more information about creating a key pair, see [How to: Create a Public/Private Key Pair](http://msdn.microsoft.com/en-us/library/6f05ezxy.aspx).

**To create and sign an assembly with a strong name using the Assembly Linker**

* At the command prompt, type the following command:

**al** **/out:**<*assembly name*> *<module name>* **/keyfile:**<*file name*>

In this command, *assembly name* is the name of the assembly to sign with a strong name, *module name* is the name of the code module used to create the assembly, and *file name* is the name of the container or file that contains the key pair.

The following example signs the assembly MyAssembly.dll with a strong name using the key file sgKey.snk.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl10_code');" \o "Copy Code)

al /out:MyAssembly.dll MyModule.netmodule /keyfile:sgKey.snk

**To sign an assembly with a strong name using attributes**

* In a code module, add the **AssemblyKeyFileAttribute** or the **AssemblyKeyNameAttribute**, specifying the name of the file or container that contains the key pair to use when signing the assembly with a strong name.

The following code example uses the **AssemblyKeyFileAttribute** with a key file called sgKey.snk, located in the directory where the assembly is compiled. This assumes that the assembly is compiled using the command-line compilers vbc.exe and csc.exe.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl11_code');" \o "Copy Code)

<Assembly:AssemblyKeyFileAttribute("sgKey.snk")>

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl12_code');" \o "Copy Code)

[assembly:AssemblyKeyFileAttribute(@"sgKey.snk")]

|  |
| --- |
| **NoteNote:** |
| In development environments such as Visual Studio, the assembly might not be compiled in the project directory. For example, some versions of Visual Studio compile C# projects in a bin\Debug subdirectory. In that case, the path in the code example would be "..\\..\\sgKey.snk". In Visual Studio 2005 the key file for C# can be specified in project settings. |

You can also delay sign an assembly when compiling. For more information, see [Delay Signing an Assembly](http://msdn.microsoft.com/en-us/library/t07a3dye.aspx).

When signing an assembly with a strong name, the [Assembly Linker (Al.exe)](http://msdn.microsoft.com/en-us/library/c405shex.aspx) looks for the key file relative to the current directory and to the output directory. When using command-line compilers, you can simply copy the key to the current directory containing your code modules.

http://i.msdn.microsoft.com/Global/Images/clear.gif

**How to: Create a Public/Private Key Pair**

To sign an assembly with a strong name, you must have a public/private key pair. This public and private cryptographic key pair is used during compilation to create a strong-named assembly. You can create a key pair using the [Strong Name tool (Sn.exe)](http://msdn.microsoft.com/en-us/library/k5b5tt23.aspx). Key pair files usually have an .snk extension.

|  |
| --- |
| **NoteNote:** |
| In Visual Studio 2005, the C#, Visual Basic, and Visual J# integrated development environments (IDEs) allow you to generate key pairs and sign assemblies without the need to create a key pair using Sn.exe. These IDEs have a **Signing** tab in the Project Designer. The Visual C++ IDE allows you to specify the location of an existing key file in the **Advanced** property page in the **Linker** section of the **Configuration Properties** sectionof the Property Pages window. The use of the [AssemblyKeyFileAttribute](http://msdn.microsoft.com/en-us/library/system.reflection.assemblykeyfileattribute.aspx) to identify key file pairs has been made obsolete in Visual Studio 2005. |

**To create a key pair**

* At the command prompt, type the following command:

**sn –k** <*file name*>

In this command, *file name* is the name of the output file containing the key pair.

The following example creates a key pair called sgKey.snk.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl04_code');" \o "Copy Code)

sn -k sgKey.snk

If you intend to delay sign an assembly and you control the whole key pair (which is unlikely outside test scenarios), you can use the following commands to generate a key pair and then extract the public key from it into a separate file. First, create the key pair:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl05_code');" \o "Copy Code)

sn -k keypair.snk

* Next, extract the public key from the key pair and copy it to a separate file:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl06_code');" \o "Copy Code)

sn -p keypair.snk public.snk

* Once you create the key pair, you must put the file where the strong name signing tools can find it.

When signing an assembly with a strong name, the [Assembly Linker (Al.exe)](http://msdn.microsoft.com/en-us/library/c405shex.aspx) looks for the key file relative to the current directory and to the output directory. When using command-line compilers, you can simply copy the key to the current directory containing your code modules.

If you are using an earlier version of Visual Studio that does not have a **Signing** tab in the project properties, the recommended key file location is the project directory with the file attribute specified as follows:

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl08_code');" \o "Copy Code)

<Assembly: AssemblyKeyFileAttribute("..\..\key.snk")>

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl09_code');" \o "Copy Code)

[assembly: AssemblyKeyFileAttribute(@"..\..\key.snk")]

**Delay Signing an Assembly**

An organization can have a closely guarded key pair that developers do not have access to on a daily basis. The public key is often available, but access to the private key is restricted to only a few individuals. When developing assemblies with strong names, each assembly that references the strong-named target assembly contains the token of the public key used to give the target assembly a strong name. This requires that the public key be available during the development process.

You can use delayed or partial signing at build time to reserve space in the portable executable (PE) file for the strong name signature, but defer the actual signing until some later stage (typically just before shipping the assembly).

The following steps outline the process to delay sign an assembly:

1. Obtain the public key portion of the key pair from the organization that will do the eventual signing. Typically this key is in the form of an .snk file, which can be created using the [Strong Name tool (Sn.exe)](http://msdn.microsoft.com/en-us/library/k5b5tt23.aspx) provided by the Windows Software Development Kit (SDK).
2. Annotate the source code for the assembly with two custom attributes from [System.Reflection](http://msdn.microsoft.com/en-us/library/system.reflection.aspx):
   * [AssemblyKeyFileAttribute](http://msdn.microsoft.com/en-us/library/system.reflection.assemblykeyfileattribute.aspx), which passes the name of the file containing the public key as a parameter to its constructor.
   * [AssemblyDelaySignAttribute](http://msdn.microsoft.com/en-us/library/system.reflection.assemblydelaysignattribute.aspx), which indicates that delay signing is being used by passing **true** as a parameter to its constructor. For example:

Visual Basic

<Assembly:AssemblyKeyFileAttribute("myKey.snk")>

<Assembly:AssemblyDelaySignAttribute(true)>

C#

[assembly:AssemblyKeyFileAttribute("myKey.snk")]

[assembly:AssemblyDelaySignAttribute(true)]

1. The compiler inserts the public key into the assembly manifest and reserves space in the PE file for the full strong name signature. The real public key must be stored while the assembly is built so that other assemblies that reference this assembly can obtain the key to store in their own assembly reference.
2. Because the assembly does not have a valid strong name signature, the verification of that signature must be turned off. You can do this by using the **–Vr** option with the Strong Name tool.

The following example turns off verification for an assembly called myAssembly.dll.

sn –Vr myAssembly.dll

|  |
| --- |
| **Caution noteCaution:** |
| Use the **-Vr** option only during development. Adding an assembly to the skip verification list creates a security vulnerability. A malicious assembly could use the fully specified assembly name (assembly name, version, culture, and public key token) of the assembly added to the skip verification list to fake its identity. This would allow the malicious assembly to also skip verification. |

1. Later, usually just before shipping, you submit the assembly to your organization's signing authority for the actual strong name signing using the **–R** option with the Strong Name tool.

The following example signs an assembly called myAssembly.dll with a strong name using the sgKey.snk key pair.

sn -R myAssembly.dll sgKey.snk

**Assembly Security Considerations**

* When you build an assembly, you can specify a set of permissions that the assembly requires to run. Whether certain permissions are granted or not granted to an assembly is based on evidence.

There are two distinct ways evidence is used:

* The input evidence is merged with the evidence gathered by the loader to create a final set of evidence used for policy resolution. The methods that use this semantic include **Assembly.Load**, **Assembly.LoadFrom**, and **Activator.CreateInstance**.
* The input evidence is used unaltered as the final set of evidence used for policy resolution. The methods that use this semantic include **Assembly.Load(byte[])** and **AppDomain.DefineDynamicAssembly()**.

Optional permissions can be granted by the [security policy](http://msdn.microsoft.com/en-us/library/33tceax8.aspx) set on the computer where the assembly will run. If you want your code to handle all potential security exceptions, you can do one of the following:

* Insert a permission request for all the permissions your code must have, and handle up front the load-time failure that occurs if the permissions are not granted.
* Do not use a permission request to obtain permissions your code might need, but be prepared to handle security exceptions if permissions are not granted.

|  |
| --- |
| **NoteNote:** |
| Security is a complex area, and you have many options to choose from. For more information, see [Key Security Concepts](http://msdn.microsoft.com/en-us/library/z164t8hs.aspx). |

At load time, the assembly's evidence is used as input to security policy. Security policy is established by the enterprise and the computer's administrator as well as by user policy settings, and determines the set of permissions that is granted to all managed code when executed. Security policy can be established for the publisher of the assembly (if it has a signing tool generated signature), for the Web site and zone (in Internet Explorer terms) the assembly was downloaded from, or for the assembly's strong name. For example, a computer's administrator can establish security policy that allows all code downloaded from a Web site and signed by a given software company to access a database on a computer, but does not grant access to write to the computer's disk.

http://i.msdn.microsoft.com/Global/Images/clear.gif Strong-Named Assemblies and Signing Tools

You can sign an assembly in two different but complementary ways: with a strong name or using either the [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) in .NET Framework version 1.0 and 1.1 or [Sign Tool (SignTool.exe)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx) in later versions of the .NET Framework. Signing an assembly with a strong name adds a public key encryption to the file containing the assembly manifest. Strong name signing helps to verify name uniqueness, prevent name spoofing, and provide callers with some identity when a reference is resolved.

However, no level of trust is associated with a strong name, which makes the [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) and [Sign Tool (SignTool.exe)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx) important. The two signing tools require a publisher to prove its identity to a third-party authority and obtain a certificate. This certificate is then embedded in your file and can be used by an administrator to decide whether to trust the code's authenticity.

You can give both a strong name and a digital signature created using the [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) or [Sign Tool (SignTool.exe)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx) to an assembly, or you can use either alone. The two signing tools can sign only one file at a time; for a multifile assembly, you sign the file that contains the assembly manifest. A strong name is stored in the file containing the assembly manifest, but a signature created using the [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) or [Sign Tool (SignTool.exe)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx) is stored in a reserved slot in the portable executable (PE) file containing the assembly manifest. Signing of an assembly using the [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) or [Sign Tool (SignTool.exe)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx) can be used (with or without a strong name) when you already have a trust hierarchy that relies on [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) or [Sign Tool (SignTool.exe)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx) generated signatures, or when your policy uses only the key portion and does not check a chain of trust.

|  |
| --- |
| **NoteNote:** |
| When using both a strong name and a signing tool signature on an assembly, the strong name must be assigned first. |

The common language runtime also performs a hash verification; the assembly manifest contains a list of all files that make up the assembly, including a hash of each file as it existed when the manifest was built. As each file is loaded, its contents are hashed and compared with the hash value stored in the manifest. If the two hashes do not match, the assembly fails to load.

Because strong naming and signing using the [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) or [Sign Tool (SignTool.exe)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx) guarantee integrity, you can base code access security policy on these two forms of assembly evidence. Strong naming and signing using the [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) or [Sign Tool (SignTool.exe)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx) guarantee integrity through digital signatures and certificates. All the technologies mentioned—hash verification, strong naming, and signing using the [File Signing Tool (Signcode.exe)](http://msdn.microsoft.com/en-us/library/9sh96ycy.aspx) or [Sign Tool (SignTool.exe)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx)—work together to ensure that the assembly has not been altered in any way.

http://i.msdn.microsoft.com/Global/Images/clear.gif

**Assembly Versioning**

All versioning of assemblies that use the common language runtime is done at the assembly level. The specific version of an assembly and the versions of dependent assemblies are recorded in the assembly's manifest. The default version policy for the runtime is that applications run only with the versions they were built and tested with, unless overridden by explicit version policy in configuration files (the application configuration file, the publisher policy file, and the computer's administrator configuration file).

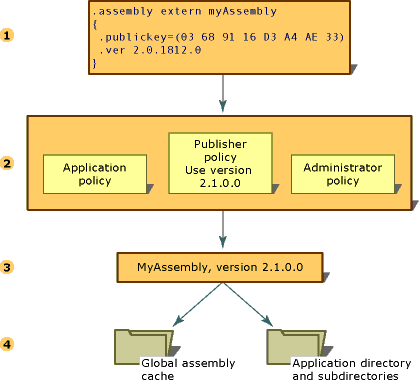
|  |
| --- |
| **NoteNote:** |
| Versioning is done only on assemblies with strong names. |

The runtime performs several steps to resolve an assembly binding request:

1. Checks the original assembly reference to determine the version of the assembly to be bound.
2. Checks for all applicable configuration files to apply version policy.
3. Determines the correct assembly from the original assembly reference and any redirection specified in the configuration files, and determines the version that should be bound to the calling assembly.
4. Checks the global assembly cache, codebases specified in configuration files, and then checks the application's directory and subdirectories using the probing rules explained in [How the Runtime Locates Assemblies](http://msdn.microsoft.com/en-us/library/yx7xezcf.aspx).

The following illustration shows these steps.

**Resolving an assembly binding request**



For more information about configuring applications, see [Configuration Files](http://msdn.microsoft.com/en-us/library/1xtk877y.aspx). For more information about binding policy, see [How the Runtime Locates Assemblies](http://msdn.microsoft.com/en-us/library/yx7xezcf.aspx).

http://i.msdn.microsoft.com/Global/Images/clear.gif Version Information

Each assembly has two distinct ways of expressing version information:

* The assembly's version number, which, together with the assembly name and culture information, is part of the assembly's identity. This number is used by the runtime to enforce version policy and plays a key part in the type resolution process at run time.
* An informational version, which is a string that represents additional version information included for informational purposes only.

**Assembly Version Number**

Each assembly has a version number as part of its identity. As such, two assemblies that differ by version number are considered by the runtime to be completely different assemblies. This version number is physically represented as a four-part string with the following format:

<*major version*>.<*minor version*>.<*build number*>.<*revision*>

For example, version 1.5.1254.0 indicates 1 as the major version, 5 as the minor version, 1254 as the build number, and 0 as the revision number.

The version number is stored in the assembly manifest along with other identity information, including the assembly name and public key, as well as information on relationships and identities of other assemblies connected with the application.

When an assembly is built, the development tool records dependency information for each assembly that is referenced in the assembly manifest. The runtime uses these version numbers, in conjunction with configuration information set by an administrator, an application, or a publisher, to load the proper version of a referenced assembly.

The runtime distinguishes between regular and strong-named assemblies for the purposes of versioning. Version checking only occurs with strong-named assemblies.

For information about specifying version binding policies, see [Configuration Files](http://msdn.microsoft.com/en-us/library/1xtk877y.aspx). For information about how the runtime uses version information to find a particular assembly, see [How the Runtime Locates Assemblies](http://msdn.microsoft.com/en-us/library/yx7xezcf.aspx).

**Assembly Informational Version**

The informational version is a string that attaches additional version information to an assembly for informational purposes only; this information is not used at run time. The text-based informational version corresponds to the product's marketing literature, packaging, or product name and is not used by the runtime. For example, an informational version could be "Common Language Runtime version 1.0" or "NET Control SP 2". On the Version tab of the file properties dialog in Microsoft Windows, this information appears in the item "Product Version".

|  |
| --- |
| **NoteNote:** |
| Although you can specify any text, a warning message appears on compilation if the string is not in the format used by the assembly version number, or if it is in that format but contains wildcards. This warning is harmless. |

The informational version is represented using the custom attribute [System.Reflection..::.AssemblyInformationalVersionAttribute](http://msdn.microsoft.com/en-us/library/system.reflection.assemblyinformationalversionattribute.aspx). For more information about the informational version attribute, see [Setting Assembly Attributes](http://msdn.microsoft.com/en-us/library/4w8c1y2s.aspx).

**Assembly Placement**

For most .NET Framework applications, you locate assemblies that make up an application in the application's directory, in a subdirectory of the application's directory, or in the global assembly cache (if the assembly is shared). You can override where the common language runtime looks for an assembly by using the [<codeBase> Element](http://msdn.microsoft.com/en-us/library/efs781xb.aspx) in a configuration file. If the assembly does not have a strong name, the location specified using the [<codeBase> Element](http://msdn.microsoft.com/en-us/library/efs781xb.aspx) is restricted to the application directory or a subdirectory. If the assembly has a strong name, the [<codeBase> Element](http://msdn.microsoft.com/en-us/library/efs781xb.aspx) can specify any location on the computer or on a network.

Similar rules apply to locating assemblies when working with unmanaged code or COM interop applications: if the assembly will be shared by multiple applications, it should be installed into the global assembly cache. Assemblies used with unmanaged code must be exported as a type library and registered. Assemblies used by COM interop must be registered in the catalog, although in some cases this registration occurs automatically.

**Assemblies and Side-by-Side Execution**

Side-by-side execution is the ability to store and execute multiple versions of an application or component on the same computer. This means that you can have multiple versions of the runtime, and multiple versions of applications and components that use a version of the runtime, on the same computer at the same time. Side-by-side execution gives you more control over what versions of a component an application binds to, and more control over what version of the runtime an application uses.

Support for side-by-side storage and execution of different versions of the same assembly is an integral part of strong naming and is built into the infrastructure of the runtime. Because the strong-named assembly's version number is part of its identity, the runtime can store multiple versions of the same assembly in the global assembly cache and load those assemblies at run time.

Although the runtime provides you with the ability to create side-by-side applications, side-by-side execution is not automatic. For more information on creating applications for side-by-side execution, see [Guidelines for Creating Applications and Components for Side-by-Side Execution](http://msdn.microsoft.com/en-us/library/z5e12zb4.aspx).

**How the Runtime Locates Assemblies**

To successfully deploy your .NET Framework application, you must understand how the common language runtime locates and binds to the assemblies that make up your application. By default, the runtime attempts to bind with the exact version of an assembly that the application was built with. This default behavior can be overridden by configuration file settings.

The common language runtime performs a number of steps when attempting to locate an assembly and resolve an assembly reference. Each step is explained in the following sections. The term probing is often used when describing how the runtime locates assemblies; it refers to the set of heuristics used to locate the assembly based on its name and culture.

|  |
| --- |
| **NoteNote:** |
| You can view binding information in the log file using the [Assembly Binding Log Viewer (Fuslogvw.exe)](http://msdn.microsoft.com/en-us/library/e74a18c4.aspx), which is included in the Windows Software Development Kit (SDK). |

Assemblies can also be loaded dynamically at run time by using reflection. [Best Practices for Assembly Loading](http://msdn.microsoft.com/en-us/library/dd153782.aspx) discusses issues that can arise when static and dynamic assembly loading interact.

 Initiating the Bind

The process of locating and binding to an assembly begins when the runtime attempts to resolve a reference to another assembly. This reference can be either static or dynamic. The compiler records static references in the assembly manifest's metadata at build time. Dynamic references are constructed on the fly as a result of calling various methods, such as [System.Reflection.Assembly.Load](http://msdn.microsoft.com/en-us/library/system.reflection.assembly.load.aspx).

The preferred way to reference an assembly is to use a full reference, including the assembly name, version, culture, and public key token (if one exists). The runtime uses this information to locate the assembly, following the steps described later in this section. The runtime uses the same resolution process regardless of whether the reference is for a static or dynamic assembly.

You can also make a dynamic reference to an assembly by providing the calling method with only partial information about the assembly, such as specifying only the assembly name. In this case, only the application directory is searched for the assembly, and no other checking occurs. You make a partial reference using any of the various methods for loading assemblies such as [System.Reflection.Assembly.Load](http://msdn.microsoft.com/en-us/library/system.reflection.assembly.load.aspx) or [System.AppDomain.Load](http://msdn.microsoft.com/en-us/library/system.appdomain.load.aspx).

Finally, you can make a dynamic reference using a method such as [System.Reflection.Assembly.Load](http://msdn.microsoft.com/en-us/library/system.reflection.assembly.load.aspx) and provide only partial information; you then qualify the reference using the [<qualifyAssembly>](http://msdn.microsoft.com/en-us/library/cd71chf0.aspx) element in the application configuration file. This element allows you to provide the full reference information (name, version, culture and, if applicable, the public key token) in your application configuration file instead of in your code. You would use this technique if you wanted to fully qualify a reference to an assembly outside the application directory, or if you wanted to reference an assembly in the global assembly cache but you wanted the convenience of specifying the full reference in the configuration file instead of in your code.

|  |
| --- |
| **NoteNote:** |
| This type of partial reference should not be used with assemblies that are shared among several applications. Because configuration settings are applied per application and not per assembly, a shared assembly using this type of partial reference would require each application using the shared assembly to have the qualifying information in its configuration file. |

The runtime uses the following steps to resolve an assembly reference:

1. [Determines the correct assembly version](http://msdn.microsoft.com/en-us/library/8f6988ab.aspx) by examining applicable configuration files, including the application configuration file, publisher policy file, and machine configuration file. If the configuration file is located on a remote machine, the runtime must locate and download the application configuration file first.
2. [Checks whether the assembly name has been bound to before](http://msdn.microsoft.com/en-us/library/aa98tba8.aspx) and, if so, uses the previously loaded assembly. If a previous request to load the assembly failed, the request fails immediately without attempting to load the assembly.

|  |
| --- |
| **NoteNote:** |
| The caching of assembly binding failures is new in the .NET Framework version 2.0. |

1. [Checks the global assembly cache](http://msdn.microsoft.com/en-us/library/4a9t8a9a.aspx). If the assembly is found there, the runtime uses this assembly.
2. [Probes for the assembly](http://msdn.microsoft.com/en-us/library/15hyw9x3.aspx) using the following steps:
   1. If configuration and publisher policy do not affect the original reference and if the bind request was created using the [Assembly..::.LoadFrom](http://msdn.microsoft.com/en-us/library/system.reflection.assembly.loadfrom.aspx) method, the runtime checks for location hints.
   2. If a codebase is found in the configuration files, the runtime checks only this location. If this probe fails, the runtime determines that the binding request failed and no other probing occurs.
   3. Probes for the assembly using the heuristics described in the [probing section](http://msdn.microsoft.com/en-us/library/15hyw9x3.aspx). If the assembly is not found after probing, the runtime requests the Windows Installer to provide the assembly. This acts as an install-on-demand feature.

|  |
| --- |
| **NoteNote:** |
| There is no version checking for assemblies without strong names, nor does the runtime check in the global assembly cache for assemblies without strong names. |

**Creating Satellite Assemblies**

The hub and spoke model described in the [Packaging and Deploying Resources](http://msdn.microsoft.com/en-us/library/sb6a8618.aspx) topic is the recommended design implementation for developing applications with resources.

The hub and spoke model requires that you place resources in specific locations, so that they can be easily located and used. If you do not compile and name resources as expected, or if you do not place them in the correct locations, the common language runtime will not be able to locate them. As a result, the runtime uses the default resource set. For more information on resource names, see [CultureInfo Class](http://msdn.microsoft.com/en-us/library/system.globalization.cultureinfo.aspx) or [Packaging and Deploying Resources](http://msdn.microsoft.com/en-us/library/sb6a8618.aspx).

 Compiling Satellite Assemblies

Use the [Assembly Linker (Al.exe)](http://msdn.microsoft.com/en-us/library/c405shex.aspx) to compile .resources files into satellite assemblies. Al.exe creates an assembly from the .resources files that you specify. By definition, satellite assemblies can only contain resources. They cannot contain any executable code.

The following Al.exe command creates a satellite assembly for the application MyApp from the file strings.de.resources.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl38_ctl00_ctl01_code');" \o "Copy Code)

al /t:lib /embed:strings.de.resources /culture:de /out:MyApp.resources.dll

The following Al.exe command also creates a satellite assembly for the application MyApp from the file strings.de.resources. The **/template** option causes the satellite assembly to inherit assembly metadata from the parent assembly MyApp.dll.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl38_ctl00_ctl02_code');" \o "Copy Code)

al /t:lib /embed:strings.de.resources /culture:de /out:MyApp.resources.dll

/template:MyApp.dll

The following table explains the Al.exe options used in these examples in more detail.

|  |  |
| --- | --- |
| **Option** | **Description** |
| **/t:**lib | The **/t** option specifies that your satellite assembly is compiled to a library (.dll ) file. A satellite assembly cannot be executed because it does not contain code and is not an application's main assembly. Therefore, you must save satellite assemblies as DLLs. |
| **/embed:**strings.de.resources | The **/embed** option specifies the name of the source file to use when Al.exe compiles the assembly. Note that you can embed multiple .resources files in a satellite assembly. However, if you are following the hub and spoke model, you must compile one satellite assembly for each culture. You can, however, create separate .resources files for strings and objects. |
| **/culture:**de | The **/culture** option specifies the culture of the resource to compile. The runtime uses this information when it searches for the resources for a specified culture. If you omit this option, Al.exe still compiles the resource, but the runtime will not be able to find it when a user requests it. |
| **/out:**MyApp.resources.dll | The **/out** option specifies the name of the output file. The name must follow the naming standard *baseName*.resources.*extension*, where the *baseName* is the name of the main assembly, and the *extension* is a viable extension (such as .dll). Note that the runtime is not able to determine the culture of a satellite assembly based on its output file name. Therefore it is important to specify a culture with the **/culture** option described above. |
| **/template:***filename* | The **/template** option specifies an assembly from which to inherit all assembly metadata, except the culture field. The assembly that the satellite assembly inherits from must have a [strong name](http://msdn.microsoft.com/en-us/library/wd40t7ad.aspx). |

For a complete list of the options available with Al.exe, see [Assembly Linker (Al.exe)](http://msdn.microsoft.com/en-us/library/c405shex.aspx).

 Compiling Satellite Assemblies With Strong Names

If you want to install satellite assemblies into the global assembly cache, they must have strong names. Strong-named assemblies are signed with a valid public/private key pair. For more information on strong names, see [Strong-Named Assemblies](http://msdn.microsoft.com/en-us/library/wd40t7ad.aspx).

When you are developing an application, it is unlikely that you will have access to the final public/private key pair. In order to install a satellite assembly in the global assembly cache and ensure that it works as expected, you can use a technique called delayed signing. When you delay sign an assembly, you reserve space in the file for the strong name signature at build time. The actual signing is delayed until a later date when the final public/private key pair is available.

**Obtaining the Public Key**

To delay sign an assembly, you must have access to the public key. You can either obtain the real public key from the organization in your company that will do the eventual signing, or create a public key using the [Strong Name Tool (Sn.exe)](http://msdn.microsoft.com/en-us/library/k5b5tt23.aspx).

The following Sn.exe command creates a test public/private key pair and saves it in the file TestKeyPair.snk. The **–k** option specifies to Sn.exe to create a new key pair and save it in the specified file.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl39_ctl00_ctl02_code');" \o "Copy Code)

sn –k TestKeyPair.snk

You can extract the public key from the file containing the test key pair. The following command extracts the public key from TestKeyPair.snk and saves it in PublicKey.snk.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl39_ctl00_ctl03_code');" \o "Copy Code)

sn –p TestKeyPair.snk PublicKey.snk

**Delay Signing an Assembly**

Once you have obtained or created the public key, use the [Assembly Linker (Al.exe)](http://msdn.microsoft.com/en-us/library/c405shex.aspx) to compile the assembly and specify delayed signing.

The following Al.exe command creates a strong-named satellite assembly for the application MyApp from the strings.ja.resources file.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl39_ctl00_ctl05_code');" \o "Copy Code)

al /t:lib /embed:strings.ja.resources /culture:ja /out:MyApp.resources.dll /delay+ /keyfile:PublicKey.snk

The **/delay+** option specifies to delay sign the assembly. The **/keyfile:** option specifies the name of the key file containing the public key to use to delay sign the assembly.

For more information on delayed signing, see [Delay Signing an Assembly](http://msdn.microsoft.com/en-us/library/t07a3dye.aspx).

Note that strong-named assemblies contain version information that the runtime uses to determine which assembly to use to meet a binding request. For more information on this topic, see [Assembly Versioning](http://msdn.microsoft.com/en-us/library/51ket42z.aspx).

**Re-signing an Assembly**

At some later date, a delay-signed satellite assembly must be re-signed with the real key pair. You can do this using Sn.exe.

The following Sn.exe command signs MyApp.resources.dll with the real key pair stored in the file RealKeyPair.snk. The **–R** option specifies to Sn.exe to re-sign a previously signed or delay-signed assembly.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl39_ctl00_ctl08_code');" \o "Copy Code)

sn –R MyApp.resources.dll RealKeyPair.snk

**Installing a Satellite Assembly in the Global Assembly Cache**

The global assembly cache is the first location the runtime searches for resources in the resource fallback process. For more information, see the "Resource Fallback Process" subtopic in the [Packaging and Deploying Resources](http://msdn.microsoft.com/en-us/library/sb6a8618.aspx) topic. Therefore, it is important to know how to install resources into the [global assembly cache](http://msdn.microsoft.com/en-us/library/yf1d93sz.aspx). A satellite assembly that you have compiled with a strong name is ready to install in the global assembly cache. You can install assemblies into the cache using the [Global Assembly Cache Tool (Gacutil.exe)](http://msdn.microsoft.com/en-us/library/ex0ss12c.aspx).

The following Gacutil.exe command installs MyApp.resources.dll into the global assembly cache.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl39_ctl00_ctl12_code');" \o "Copy Code)

gacutil /i:MyApp.resources.dll

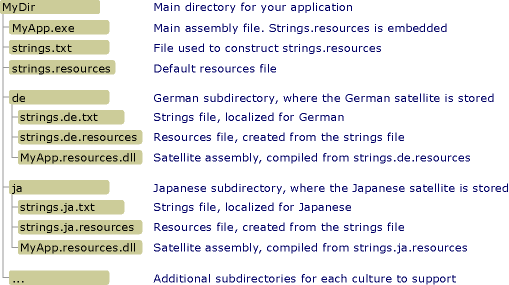
The **/i** option specifies to Gacutil.exe to install the specified assembly into the global assembly cache. As a result of this command, an entry is placed in the cache, which allows entries in this .resources file to be accessed. After being installed in the cache, the specified resource is available to all applications that are designed to use it.

 Directory Locations for Satellite Assemblies Not Installed in the Global Assembly Cache

After you have compiled your satellite assemblies, they all have the same name. The runtime differentiates between them based upon the culture specified at compile time with Al.exe's **/culture** option and by each assembly's directory location. You must place your satellite assemblies in expected directory locations.

The following illustration shows a sample directory structure and location requirements for applications that you are not installing in the [global assembly cache](http://msdn.microsoft.com/en-us/library/yf1d93sz.aspx). The items with .txt and .resources file extensions will not ship with the final application. These are the intermediate resource files used to create the final satellite resource assemblies. In this example, you could substitute .resx files for the .txt files. The .resx files are the only type of intermediate resource file that can contain objects.

**Satellite assembly directory**



|  |
| --- |
| **NoteNote:** |
| If your application includes resources for subcultures, place each subculture in its own directory. Do not place subcultures in subdirectories of their main culture's directory. |