.NET Framework 4

**Assemblies in the Common Language Runtime**

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.

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 that is loaded in the scope of one assembly is not the same as a type called MyType that is 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.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifRelated Topics

|  |  |
| --- | --- |
| **Title** | **Description** |
| [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. |
| [Programming with Assemblies](http://msdn.microsoft.com/en-us/library/8wxf689z.aspx) | Describes how to create, sign, and set attributes on assemblies. |
| [Emitting Dynamic Methods and Assemblies](http://msdn.microsoft.com/en-us/library/8ffc3x75.aspx) | Describes how to create dynamic assemblies. |
| [How the Runtime Locates Assemblies](http://msdn.microsoft.com/en-us/library/yx7xezcf.aspx) | Describes how the .NET Framework resolves assembly references at run time. |

.NET Framework 4

**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.

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.

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.

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).

.NET Framework 4

**Assembly Contents**

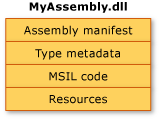
In general, a static assembly can consist of four elements:

* 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.

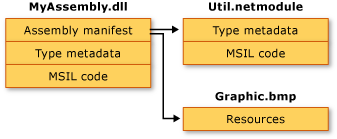
**Single-file assembly**



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.

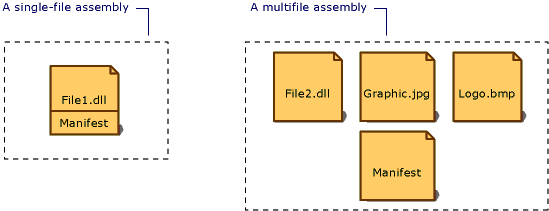
.NET Framework 4

**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.

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).

.NET Framework 4

**Global Assembly Cache**

Updated: June 2010

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 two 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).

|  |
| --- |
| **NoteNote** |
| In deployment scenarios, use Windows Installer 2.0 to install assemblies into the global assembly cache. Use the Global Assembly Cache tool only in development scenarios, because it does 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.

.NET Framework 4

**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.

.NET Framework 4

**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.

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 [SignTool.exe (Sign Tool)](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 [SignTool.exe (Sign Tool)](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 [SignTool.exe (Sign Tool)](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 [SignTool.exe (Sign Tool)](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 [SignTool.exe (Sign Tool)](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 [SignTool.exe (Sign Tool)](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 [SignTool.exe (Sign Tool)](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 [SignTool.exe (Sign Tool)](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 [SignTool.exe (Sign Tool)](http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx)—work together to ensure that the assembly has not been altered in any way.

.NET Framework 4

**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.

.NET Framework 4

**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).

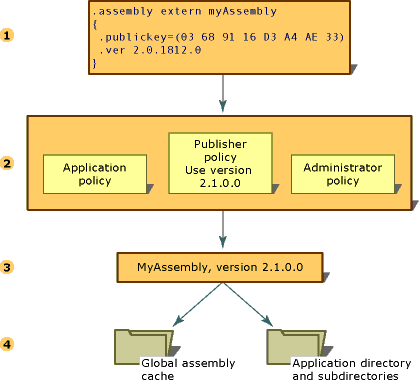
|  |
| --- |
| **NoteNote** |
| Versioning is done only on assemblies with strong names. |

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).

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).

.NET Framework 4

**Guidelines for Creating Applications and Components for Side-by-Side Execution**

Follow these general guidelines to create managed applications or components designed for side-by-side execution:

* Bind type identity to a particular version of a file.

The common language runtime binds type identity to a particular file version by using strong-named assemblies. To create an application or component for side-by-side execution, you must give all assemblies a strong name. This creates precise type identity and ensures that any type resolution is directed to the correct file. A strong-named assembly contains version, culture, and publisher information that the runtime uses to locate the correct file to fulfill a binding request.

* Use version-aware storage.

The runtime uses the global assembly cache to provide version-aware storage. The global assembly cache is a version-aware directory structure installed on every computer that uses the .NET Framework. Assemblies installed in the global assembly cache are not overwritten when a new version of that assembly is installed.

* Create an application or component that runs in isolation.

An application or component that runs in isolation must manage resources to avoid conflicts when two instances of the application or component are running simultaneously. The application or component must also use a version-specific file structure.

Application and Component Isolation

One key to successfully designing an application or component for side-by-side execution is isolation. The application or component must manage all resources, particularly file I/O, in an isolated manner. Follow these guidelines to make sure your application or component runs in isolation:

* Write to the registry in a version-specific way. Store values in hives or keys that indicate the version, and do not share information or state across versions of a component. This prevents two applications or components running at the same time from overwriting information.
* Make named kernel objects version-specific so that a race condition does not occur. For example, a race condition occurs when two semaphores from two versions of the same application wait on each other.
* Make file and directory names version-aware. This means that file structures should rely on version information.
* Create user accounts and groups in a version-specific manner. User accounts and groups created by an application should be identified by version. Do not share user accounts and groups between versions of an application.

Installing and Uninstalling Versions

When designing an application for side-by-side execution, follow these guidelines concerning installing and uninstalling versions:

* Do not delete information from the registry that may be needed by other applications running under a different version of the .NET Framework.
* Do not replace information in the registry that may be needed by other applications running under a different version of the .NET Framework.
* Do not unregister COM components that may be needed by other applications running under a different version of the .NET Framework.
* Do not change **InprocServer32** or other registry entries for a COM server that was already registered.
* Do not delete user accounts or groups that may be needed by other applications running under a different version of the .NET Framework.
* Do not add anything to the registry that contains an unversioned path.

File Version Number and Assembly Version Number

File version is a Win32 version resource that is not used by the runtime. In general, you update the file version even for an in-place QFE. Two identical files can have different file version information, and two different files can have the same file version information.

The assembly version is used by the runtime for assembly binding. Two identical assemblies with different version numbers are treated as two different assemblies by the runtime.

The [Global Assembly Cache tool (Gacutil.exe)](http://msdn.microsoft.com/en-us/library/ex0ss12c.aspx) allows you to replace an assembly when only the file version number is newer. The installer generally does not install over an assembly unless the assembly version number is greater.

.NET Framework 4

**Creating Assemblies**

You can create single-file or multifile assemblies using an IDE, such as Visual Studio 2005, or the compilers and tools provided by the Windows Software Development Kit (SDK). The simplest assembly is a single file that has a simple name and is loaded into a single application domain. This assembly cannot be referenced by other assemblies outside the application directory and does not undergo version checking. To uninstall the application made up of the assembly, you simply delete the directory where it resides. For many developers, an assembly with these features is all that is needed to deploy an application.

You can create a multifile assembly from several code modules and resource files. You can also create an assembly that can be shared by multiple applications. A shared assembly must have a strong name and can be deployed in the global assembly cache.

You have several options when grouping code modules and resources into assemblies, depending on the following factors:

* Versioning

Group modules that should have the same version information.

* Deployment

Group code modules and resources that support your model of deployment.

* Reuse

Group modules if they can be logically used together for some purpose. For example, an assembly consisting of types and classes used infrequently for program maintenance can be put in the same assembly. In addition, types that you intend to share with multiple applications should be grouped into an assembly and the assembly should be signed with a strong name.

* Security

Group modules containing types that require the same security permissions.

* Scoping

Group modules containing types whose visibility should be restricted to the same assembly.

Special considerations must be made when making common language runtime assemblies available to unmanaged COM applications. For more information about working with unmanaged code, see [Exposing .NET Framework Components to COM](http://msdn.microsoft.com/en-us/library/zsfww439.aspx).

.NET Framework 4

**How to: Build a Single-File Assembly**

A single-file assembly, which is the simplest type of assembly, contains type information and implementation, as well as the [assembly manifest](http://msdn.microsoft.com/en-us/library/1w45z383.aspx). You can use command-line compilers or Visual Studio 2005 to create a single-file assembly. By default, the compiler creates an assembly file with an .exe extension.

|  |
| --- |
| **NoteNote** |
| Visual Studio 2005 for C# and Visual Basic can be used only to create single-file assemblies. If you want to create multifile assemblies, you must use command-line compilers or Visual Studio 2005 for Visual C++. |

The following procedures show how to create single-file assemblies using command-line compilers.

### To create an assembly with an .exe extension

* At the command prompt, type the following command:

<compiler command> <module name>

In this command, compiler command is the compiler command for the language used in your code module, and module name is the name of the code module to compile into the assembly.

The following example creates an assembly named myCode.exe from a code module called myCode.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl03_code');" \o "Copy Code)

csc myCode.cs

Visual Basic

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

vbc myCode.vb

### To create an assembly with an .exe extension and specify the output file name

* At the command prompt, type the following command:

<compiler command> **/out:**<file name> <module name>

In this command, compiler command is the compiler command for the language used in your code module, file name is the output file name, and module name is the name of the code module to compile into the assembly.

The following example creates an assembly named myAssembly.exe from a code module called myCode.

C#

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

csc /out:myAssembly.exe myCode.cs

Visual Basic

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

vbc /out:myAssembly.exe myCode.vb

Creating Library Assemblies

A library assembly is similar to a class library. It contains types that will be referenced by other assemblies, but it has no entry point to begin execution.

### To create a library assembly

* At the command prompt, type the following command:

<compiler command> **/t:library** <module name>

In this command, compiler command is the compiler command for the language used in your code module, and module name is the name of the code module to compile into the assembly. You can also use other compiler options, such as the **/out:** option.

The following example creates a library assembly named myCodeAssembly.dll from a code module called myCode.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl20_ctl00_ctl00_code');" \o "Copy Code)

csc /out:myCodeLibrary.dll /t:library myCode.cs

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl20_ctl00_ctl01_code');" \o "Copy Code)

vbc /out:myCodeLibrary.dll /t:library myCode.vb

.NET Framework 4

**How to: Build a Multifile Assembly**

This section describes the procedure used to create a multifile assembly and provides a complete example that illustrates each of the steps in the procedure.

### To create a multifile assembly

1. Compile all files that contain namespaces referenced by other modules in the assembly into code modules. The default extension for code modules is .netmodule. For example, if a file called Stringer creates a namespace called myStringer that is referenced in the Client file code, Stringer should be compiled into a code module first.
2. Compile all other modules, using the necessary compiler options to indicate the other modules that are referenced in the code.
3. Use the [Assembly Linker (Al.exe)](http://msdn.microsoft.com/en-us/library/c405shex.aspx) to create the output file that contains the assembly manifest. This file contains reference information for all modules or resources that are part of the assembly.

|  |
| --- |
| **NoteNote** |
| The Visual Studio 2005 IDE for C# and Visual Basic can only be used to create single-file assemblies. If you want to create multifile assemblies, you must use the command-line compilers or Visual Studio 2005 with Visual C++. |

The following example illustrates step 1 of the procedure above, by compiling files with namespaces referenced by other files. This example starts with some simple code for the Stringer file. Stringer has a namespace called myStringer with a class called Stringer. The Stringer class contains a method called StringerMethod that writes a single line to the console.

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl03_code');" \o "Copy Code)

' Assembly building example in the .NET Framework.

Imports System

Namespace myStringer

Public Class Stringer

Public Sub StringerMethod()

System.Console.WriteLine("This is a line from StringerMethod.")

End Sub

End Class

End Namespace

C#

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

// Assembly building example in the .NET Framework.

using System;

namespace myStringer

{

public class Stringer

{

public void StringerMethod()

{

System.Console.WriteLine("This is a line from StringerMethod.");

}

}

}

Visual C++

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

// Assembly building example in the .NET Framework.

using namespace System;

namespace myStringer

{

public ref class Stringer

{

public:

void StringerMethod()

{

System::Console::WriteLine("This is a line from StringerMethod.");

}

};

}

Use the following command to compile this code:

Visual Basic

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

vbc /t:module Stringer.vb

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl07_code');" \o "Copy Code)

csc /t:module Stringer.cs

Visual C++

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

cl /clr:pure /LN Stringer.cpp

Specifying the module parameter with the **/t:** compiler option indicates that the file should be compiled as a module rather than as an assembly. The compiler produces a module called Stringer.netmodule, which can be added to an assembly.

In step two of the procedure above, you must compile modules with references to other modules. This step uses the **/addmodule** compiler option. In the following example, a code module called Client has an entry point Main method that references a method in the Stringer.dll module created in Step 1.

The following example shows the code for Client.

Visual Basic

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

Imports System

Imports myStringer 'The namespace created in Stringer.netmodule.

Class MainClientApp

' Static method Main is the entry point method.

Public Shared Sub Main()

Dim myStringInstance As New Stringer()

Console.WriteLine("Client code executes")

myStringInstance.StringerMethod()

End Sub

End Class

C#

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

using System;

using myStringer; //The namespace created in Stringer.netmodule.

class MainClientApp

{

// Static method Main is the entry point method.

public static void Main()

{

Stringer myStringInstance = new Stringer();

Console.WriteLine("Client code executes");

myStringInstance.StringerMethod();

}

}

Visual C++

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

#using "Stringer.netmodule"

using namespace System;

using namespace myStringer; //The namespace created in Stringer.netmodule.

ref class MainClientApp

{

// Static method Main is the entry point method.

public:

static void Main()

{

Stringer^ myStringInstance = gcnew Stringer();

Console::WriteLine("Client code executes");

myStringInstance->StringerMethod();

}

};

int main()

{

MainClientApp::Main();

}

Use the following command to compile this code:

Visual Basic

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

vbc /addmodule:Stringer.netmodule /t:module Client.vb

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl13_code');" \o "Copy Code)

csc /addmodule:Stringer.netmodule /t:module Client.cs

Visual C++

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl14_code');" \o "Copy Code)

cl /clr:pure /FUStringer.netmodule /LN Client.cpp

Specify the **/t:module** option because this module will be added to an assembly in a future step. Specify the **/addmodule** option because the code in Client references a namespace created by the code in Stringer.netmodule. The compiler produces a module called Client.netmodule that contains a reference to another module, Stringer.netmodule.

|  |
| --- |
| **NoteNote** |
| The C# and Visual Basic compilers support directly creating multifile assemblies using the following two different syntaxes. |

* Two compilations create a two-file assembly:

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl16_code');" \o "Copy Code)

vbc /t:module Stringer.vb

vbc Client.vb /addmodule:Stringer.netmodule

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl17_code');" \o "Copy Code)

csc /t:module Stringer.cs

csc Client.cs /addmodule:Stringer.netmodule

Visual C++

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl18_code');" \o "Copy Code)

cl /clr:pure /LN Stringer.cpp

cl /clr:pure Client.cpp /link /ASSEMBLYMODULE:Stringer.netmodule

* One compilation creates a two-file assembly:

Visual Basic

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl19_code');" \o "Copy Code)

vbc /out:Client.exe Client.vb /out:Stringer.netmodule Stringer.vb

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl20_code');" \o "Copy Code)

csc /out:Client.exe Client.cs /out:Stringer.netmodule Stringer.cs

Visual C++

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl21_code');" \o "Copy Code)

cl /clr:pure /LN Stringer.cpp

cl /clr:pure Client.cpp /link /ASSEMBLYMODULE:Stringer.netmodule

You can use the [Assembly Linker (Al.exe)](http://msdn.microsoft.com/en-us/library/c405shex.aspx) to create an assembly from a collection of compiled code modules.

### To create a multifile assembly using the Assembly Linker

* At the command prompt, type the following command:

**al** <module name> <module name> … **/main:**<method name> **/out:**<file name> **/target:**<assembly file type>

In this command, the module name arguments specify the name of each module to include in the assembly. The **/main:** option specifies the method name that is the assembly's entry point. The **/out:** option specifies the name of the output file, which contains assembly metadata. The **/target:** option specifies that the assembly is a console application executable (.exe) file, a Windows executable (.win) file, or a library (.lib) file.

In the following example, Al.exe creates an assembly that is a console application executable called myAssembly.exe. The application consists of two modules called Client.netmodule and Stringer.netmodule, and the executable file called myAssembly.exe, which contains only assembly metadata . The entry point of the assembly is the Main method in the class MainClientApp, which is located in Client.dll.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl23_code');" \o "Copy Code)

al Client.netmodule Stringer.netmodule /main:MainClientApp.Main /out:myAssembly.exe /target:exe

You can use the [MSIL Disassembler (Ildasm.exe)](http://msdn.microsoft.com/en-us/library/f7dy01k1.aspx) to examine the contents of an assembly, or determine whether a file is an assembly or a module.

.NET Framework 4

**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

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

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

<Assembly:AssemblyDelaySignAttribute(True)>

C#

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

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

[assembly:AssemblyDelaySignAttribute(true)]

Visual C++

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl07_code');" \o "Copy Code)

[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.

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

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. |

|  |
| --- |
| **NoteNote** |
| If you use delay signing during development with Visual Studio on a 64-bit computer, and you compile an assembly for **Any CPU**, you might have to apply the **-Vr** option twice. (In Visual Studio, **Any CPU** is a value of the **Platform Target** build property; when you compile from the command line, it is the default.) To run your application from the command line or from Windows Explorer, use the 64-bit version of the [Sn.exe (Strong Name Tool)](http://msdn.microsoft.com/en-us/library/k5b5tt23.aspx) to apply the **-Vr** option to the assembly. To load the assembly into Visual Studio at design time (for example, if the assembly contains components that are used by other assemblies in your application), use the 32-bit version of the strong-name tool. This is because the just-in-time (JIT) compiler compiles the assembly to 64-bit native code when the assembly is run from the command line, and to 32-bit native code when the assembly is loaded into the design-time environment. |

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.

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

sn -R myAssembly.dll sgKey.snk

.NET Framework 4

**Emitting Dynamic Methods and Assemblies**

This section describes a set of managed types in the [System.Reflection.Emit](http://msdn.microsoft.com/en-us/library/system.reflection.emit.aspx) namespace that allow a compiler or tool to emit metadata and Microsoft intermediate language (MSIL) at run time and optionally generate a portable executable (PE) file on disk. Script engines and compilers are the primary users of this namespace. In this section, the functionality provided by the [System.Reflection.Emit](http://msdn.microsoft.com/en-us/library/system.reflection.emit.aspx) namespace is referred to as reflection emit.

Reflection emit provides the following capabilities:

* Define lightweight global methods at run time, using the [DynamicMethod](http://msdn.microsoft.com/en-us/library/system.reflection.emit.dynamicmethod.aspx) class, and execute them using delegates.
* Define assemblies at run time and then run them and/or save them to disk.
* Define assemblies at run time, run them, and then unload them and allow garbage collection to reclaim their resources.
* Define modules in new assemblies at run time and then run and/or save them to disk.
* Define types in modules at run time, create instances of these types, and invoke their methods.
* Define symbolic information for defined modules that can be used by tools such as debuggers and code profilers.

In addition to the managed types in the [System.Reflection.Emit](http://msdn.microsoft.com/en-us/library/system.reflection.emit.aspx) namespace, there are unmanaged metadata interfaces which are described in the [Metadata Interfaces](http://msdn.microsoft.com/en-us/library/ms233411.aspx) reference documentation. Managed reflection emit provides stronger semantic error checking and a higher level of abstraction of the metadata than the unmanaged metadata interfaces.

Another useful resource for working with metadata and MSIL is the Common Language Infrastructure (CLI) documentation, especially "Partition II: Metadata Definition and Semantics" and "Partition III: CIL Instruction Set". The documentation is available online on [MSDN](http://go.microsoft.com/fwlink/?LinkID=65555) and at the [Ecma Web site](http://go.microsoft.com/fwlink/?LinkId=116487).

In This Section

[Reflection Emit Abstractions](http://msdn.microsoft.com/en-us/library/6dc8142t.aspx)

Lists and describes the reflection emit classes you can use to define assemblies and types, methods and method parameters, events, constructors, properties, and so forth.

[Collectible Assemblies for Dynamic Type Generation](http://msdn.microsoft.com/en-us/library/dd554932.aspx)

Describes the creation and use of dynamic assemblies that can be unloaded and reclaimed by garbage collection when none of the types they contain are reachable by executing code.

[Reflection Emit Dynamic Method Scenarios](http://msdn.microsoft.com/en-us/library/sfk2s47t.aspx)

Describes the definition and use of lightweight dynamic methods.

[Reflection Emit Dynamic Assembly Scenarios](http://msdn.microsoft.com/en-us/library/tt9483fk.aspx)

Describes the core reflection emit scenario, emitting a dynamic module defined in a dynamic assembly.

[Reflection Emit Application Scenarios](http://msdn.microsoft.com/en-us/library/a6x89439.aspx)

Describes the application scenarios supported by reflection emit, including executing a script in a Web browser, executing a script in an ASP.NET page, and compiling a regular expression.

[Security Issues in Reflection Emit](http://msdn.microsoft.com/en-us/library/9syytdak.aspx)

Describes security issues related to creating dynamic assemblies using reflection emit.

[Using Reflection Emit](http://msdn.microsoft.com/en-us/library/3y322t50.aspx)

Describes fundamental tasks you can perform using reflection emit, with how-to topics on generating lightweight dynamic methods and emitting generic types and methods.

[More Information About Reflection Emit](http://msdn.microsoft.com/en-us/library/07deyh8x.aspx)

Describes other documentation related to reflection emit.

Reference

[OpCodes](http://msdn.microsoft.com/en-us/library/system.reflection.emit.opcodes.aspx)

Catalogs the MSIL instruction codes you can use to build method bodies.

[System.Reflection.Emit](http://msdn.microsoft.com/en-us/library/system.reflection.emit.aspx)

Contains managed classes used to emit dynamic methods, assemblies, and types.

[Type](http://msdn.microsoft.com/en-us/library/system.type.aspx)

Describes the [Type](http://msdn.microsoft.com/en-us/library/system.type.aspx) class, which represents types in managed reflection and reflection emit, and which is key to the use of these technologies.

[System.Reflection](http://msdn.microsoft.com/en-us/library/system.reflection.aspx)

Contains managed classes used to explore metadata and managed code.

.NET Framework 4

**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). |

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 is failed 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. |