.NET Framework 4

**Compiling MSIL to Native Code**

[This documentation is for preview only, and is subject to change in later releases. Blank topics are included as placeholders.]

Before you can run Microsoft intermediate language (MSIL), it must be compiled against the common language runtime to native code for the target machine architecture. The .NET Framework provides two ways to perform this conversion:

* A .NET Framework just-in-time (JIT) compiler.
* The .NET Framework [Ngen.exe (Native Image Generator)](http://msdn.microsoft.com/en-us/library/6t9t5wcf.aspx).

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifCompilation by the Just-in-time Compiler

JIT compilation converts MSIL to native code on demand at application run time, when the contents of an assembly are loaded and executed. Because the common language runtime supplies a JIT compiler for each supported CPU architecture, developers can build a set of MSIL assemblies that can be JIT-compiled and run on different computers with different machine architectures. However, your managed code will run only on a specific operating system if it calls platform-specific native APIs, or a platform-specific class library.

JIT compilation takes into account the fact that some code might never get called during execution. Rather than using time and memory to convert all the MSIL in a portable executable (PE) file to native code, it converts the MSIL as needed during execution and stores the resulting native code in memory so that it is accessible for subsequent calls in the context of that process. The loader creates and attaches a stub to each method in a type when the type is loaded and initialized. When a method is called for the first time, the stub passes control to the JIT compiler, which converts the MSIL for that method into native code and modifies the stub to point directly to the generated native code. Subsequent calls to the JIT-compiled method therefore proceed directly to the native code.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifInstall-Time Code Generation Using NGen.exe

Because the JIT compiler converts an assembly's MSIL to native code when individual methods defined in that assembly are called, it necessarily involves a performance hit at run time. In most cases, that performance hit is acceptable. More importantly, the code generated by the JIT compiler is bound to the process that triggered the compilation. It cannot be shared across multiple processes. To allow the generated code to be shared across multiple invocations of an application or across multiple processes that share a set of assemblies, the common language runtime supports an ahead-of-time compilation mode. This ahead-of-time compilation mode uses the [Ngen.exe (Native Image Generator)](http://msdn.microsoft.com/en-us/library/6t9t5wcf.aspx) to convert MSIL assemblies to native code much like the JIT compiler does. However, the operation of Ngen.exe differs from that of the JIT compiler in three ways:

* It performs the conversion from MSIL to native code before rather than while running the application.
* It compiles an entire assembly at a time, rather than a method at a time.
* It persists the generated code in the Native Image Cache as a file on disk.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifCode Verification

As part of compiling MSIL to native code, the MSIL code must pass a verification process unless an administrator has established a security policy that allows the code to bypass verification. Verification examines MSIL and metadata to find out whether the code is type safe, which means that it only accesses the memory locations it is authorized to access. Type safety helps isolate objects from each other and therefore helps protect them from inadvertent or malicious corruption. It also provides assurance that security restrictions on code can be reliably enforced.

The runtime relies on the fact that the following statements are true for code that is verifiably type safe:

* A reference to a type is strictly compatible with the type being referenced.
* Only appropriately defined operations are invoked on an object.
* Identities are what they claim to be.

During the verification process, MSIL code is examined in an attempt to confirm that the code can access memory locations and call methods only through properly defined types. For example, code cannot allow an object's fields to be accessed in a manner that allows memory locations to be overrun. Additionally, verification inspects code to determine whether the MSIL has been correctly generated, because incorrect MSIL can lead to a violation of the type safety rules. The verification process passes a well-defined set of type-safe code, and it passes only code that is type safe. However, some type-safe code might not pass verification because of some limitations of the verification process, and some languages, by design, do not produce verifiably type-safe code. If type-safe code is required by the security policy but the code does not pass verification, an exception is thrown when the code is run.

.NET Framework 4

**Ngen.exe (Native Image Generator)**

Updated: March 2012

The Native Image Generator (Ngen.exe) is a tool that improves the performance of managed applications. Ngen.exe creates native images, which are files containing compiled processor-specific machine code, and installs them into the native image cache on the local computer. The runtime can use native images from the cache instead of using the just-in-time (JIT) compiler to compile the original assembly.

Changes to Ngen.exe in the .NET Framework version 4:

* Ngen.exe now compiles assemblies with full trust, and code access security (CAS) policy is no longer evaluated.
* Native images that are generated with Ngen.exe can no longer be loaded into applications that are running in partial trust.

Changes to Ngen.exe in the .NET Framework version 2.0:

* Installing an assembly also installs its dependencies, simplifying the syntax of Ngen.exe.
* Native images can now be shared across application domains.
* A new action, update, re-creates images that have been invalidated.
* Actions can be deferred for execution by a service that uses idle time on the computer to generate and install images.
* Some causes of image invalidation have been eliminated.

For additional information on using Ngen.exe and the native image service, see [Native Image Service](http://msdn.microsoft.com/en-us/library/ms165074.aspx).

|  |
| --- |
| **Description: NoteNote** |
| Ngen.exe syntax for versions 1.0 and 1.1 of the .NET Framework can be found in [Native Image Generator (Ngen.exe) Legacy Syntax](http://msdn.microsoft.com/en-us/library/ms165073.aspx). |

This tool is installed with the .NET Framework redistributable package. To run the tool, we recommend that you use the Visual Studio Command Prompt or the Windows SDK Command Prompt (CMD Shell). These utilities enable you to run the tool easily, without navigating to the installation folder. For more information, see [Visual Studio and Windows SDK Command Prompts](http://msdn.microsoft.com/en-us/library/ms229859.aspx).

* If you have Visual Studio installed on your computer: On the taskbar, click Start, click All Programs, click Visual Studio, click Visual Studio Tools, and then click Visual Studio Command Prompt.

-or-

If you have the Windows SDK installed on your computer: On the taskbar, click Start, click All Programs, click the folder for the Windows SDK, and then click Command Prompt (or CMD Shell).

* At the command prompt, type the following:

ngen action [options]

ngen /? | /help

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

The following table shows the syntax of each action. For descriptions of the individual parts of an action, see the [Arguments](http://msdn.microsoft.com/en-us/library/6t9t5wcf.aspx#ArgumentTable), [Priority Levels](http://msdn.microsoft.com/en-us/library/6t9t5wcf.aspx#PriorityTable), [Scenarios](http://msdn.microsoft.com/en-us/library/6t9t5wcf.aspx#ScenarioTable), and [Config](http://msdn.microsoft.com/en-us/library/6t9t5wcf.aspx#ConfigTable) tables. The [Options](http://msdn.microsoft.com/en-us/library/6t9t5wcf.aspx#OptionTable) table describes the options and the help switches.

|  |  |
| --- | --- |
| **Action** | **Description** |
| install [assemblyName | assemblyPath] [scenarios] [config] [/queue[:{1|2|3}]] | Generate native images for an assembly and its dependencies and install the images in the native image cache.  If /queue is specified, the action is queued for the native image service. The default priority is 3. See the [Priority Levels](http://msdn.microsoft.com/en-us/library/6t9t5wcf.aspx#PriorityTable) table. |
| uninstall [assemblyName | assemblyPath] [scenarios] [config] | Delete the native images of an assembly and its dependencies from the native image cache.  To uninstall a single image and its dependencies, use the same command-line arguments that were used to install the image.  Description: Note**Note**  In the .NET Framework version 4, the action uninstall \* is no longer supported. |
| update [/queue] | Update native images that have become invalid.  If /queue is specified, the updates are queued for the native image service. Updates are always scheduled at priority 3, so they run when the computer is idle. |
| display [assemblyName | assemblyPath] | Display the state of the native images for an assembly and its dependencies.  If no argument is supplied, everything in the native image cache is displayed. |
| executeQueuedItems [1|2|3]  -or-  eqi [1|2|3] | Execute queued compilation jobs.  If a priority is specified, compilation jobs with greater or equal priority are executed. If no priority is specified, all queued compilation jobs are executed. |
| queue {pause | continue | status} | Pause the native image service, allow the paused service to continue, or query the status of the service. |

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

|  |  |
| --- | --- |
| **Argument** | **Description** |
| assemblyName | The full display name of the assembly. For example, "myAssembly, Version=2.0.0.0, Culture=neutral, PublicKeyToken=0038abc9deabfle5".  Description: Note**Note**  You can supply a partial assembly name, such as myAssembly, for the display and uninstall actions.  Only one assembly can be specified per Ngen.exe command line. |
| assemblyPath | The explicit path of the assembly. You can specify a full or relative path.  If you specify a file name without a path, the assembly must be located in the current directory.  Only one assembly can be specified per Ngen.exe command line. |

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifPriority Levels

|  |  |
| --- | --- |
| **Priority** | **Description** |
| 1 | Native images are generated and installed immediately, without waiting for idle time. |
| 2 | Native images are generated and installed without waiting for idle time, but after all priority 1 actions (and their dependencies) have completed. |
| 3 | Native images are installed when the native image service detects that the computer is idle. See [Native Image Service](http://msdn.microsoft.com/en-us/library/ms165074.aspx). |

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

|  |  |
| --- | --- |
| **Scenario** | **Description** |
| /Debug | Generate native images that can be used under a debugger. |
| /Profile | Generate native images that can be used under a profiler. |
| /NoDependencies | Generate the minimum number of native images required by the specified scenario options. |

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

|  |  |
| --- | --- |
| **Configuration** | **Description** |
| /ExeConfig:exePath | Use the configuration of the specified executable assembly.  Ngen.exe needs to make the same decisions as the loader when binding to dependencies. When a shared component is loaded at run time, using the [Load](http://msdn.microsoft.com/en-us/library/system.reflection.assembly.load.aspx) method, the application's configuration file determines the dependencies that are loaded for the shared component — for example, the version of a dependency that is loaded. The /ExeConfig switch gives Ngen.exe guidance on which dependencies would be loaded at run time. |
| /AppBase:directoryPath | When locating dependencies, use the specified directory as the application base. |

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

|  |  |
| --- | --- |
| **Option** | **Description** |
| /nologo | Suppress the Microsoft startup banner display. |
| /silent | Suppress the display of success messages. |
| /verbose | Display detailed information for debugging.  Description: Note**Note**  Due to operating system limitations, this option does not display as much additional information on Windows 98 and Windows Millennium Edition. |
| /help, /? | Display command syntax and options for the current release. |

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

To run Ngen.exe, you must have administrative privileges.

|  |
| --- |
| **Description: Caution noteCaution** |
| Do not run Ngen.exe on assemblies that are not fully trusted. Starting with the .NET Framework version 4, Ngen.exe compiles assemblies with full trust, and code access security (CAS) policy is no longer evaluated. |

Starting with the .NET Framework 4, the native images that are generated with Ngen.exe can no longer be loaded into applications that are running in partial trust. Instead, the just-in-time (JIT) compiler is invoked.

Ngen.exe generates native images for the specified assembly and all its dependencies. Dependencies are determined from references in the assembly manifest. The only scenario in which you need to install a dependency separately is when the application loads it using reflection, for example by calling the [Assembly.Load](http://msdn.microsoft.com/en-us/library/system.reflection.assembly.load.aspx) method.

|  |
| --- |
| **Description: Important noteImportant** |
| Do not use the [Assembly.LoadFrom](http://msdn.microsoft.com/en-us/library/system.reflection.assembly.loadfrom.aspx) method with native images. An image loaded with this method cannot be used by other assemblies in the execution context. |

Ngen.exe maintains a count on dependencies. For example, suppose MyAssembly.exe and YourAssembly.exe are both installed in the native image cache, and both have references to OurDependency.dll. If MyAssembly.exe is uninstalled, OurDependency.dll is not uninstalled. It is only removed when YourAssembly.exe is also uninstalled.

If you are generating a native image for an assembly in the global assembly cache, specify its display name. See [Assembly.FullName](http://msdn.microsoft.com/en-us/library/system.reflection.assembly.fullname.aspx).

The native images that Ngen.exe generates can be shared across application domains. This means you can use Ngen.exe in application scenarios that require assemblies to be shared across application domains. To specify domain neutrality:

* Apply the [LoaderOptimizationAttribute](http://msdn.microsoft.com/en-us/library/system.loaderoptimizationattribute.aspx) attribute to your application.
* Set the [AppDomainSetup.LoaderOptimization](http://msdn.microsoft.com/en-us/library/system.appdomainsetup.loaderoptimization.aspx) property when you create setup information for a new application domain.

Always use domain-neutral code when loading the same assembly into multiple application domains. If a native image is loaded into a nonshared application domain after having been loaded into a shared domain, it cannot be used.

|  |
| --- |
| **Description: NoteNote** |
| Domain-neutral code cannot be unloaded, and performance may be slightly slower, particularly when accessing static members. |

### Generating Images for Different Scenarios

After you have generated a native image for an assembly, the runtime automatically attempts to locate and use this native image each time it runs the assembly. Multiple images can be generated, depending on usage scenarios.

For example, if you run an assembly in a debugging or profiling scenario, the runtime looks for a native image that was generated with the /Debug or /Profile options. If it is unable to find a matching native image, the runtime reverts to standard JIT compilation. The only way to debug native images is to create a native image with the /Debug option.

The uninstall action also recognize scenarios, so you can uninstall all scenarios or only selected scenarios.

### Determining When to Use Native Images

Native images can provide performance improvements in two areas: improved memory use and reduced startup time.

|  |
| --- |
| **Description: NoteNote** |
| Performance of native images depends on a number of factors that make analysis difficult, such as code and data access patterns, how many calls are made across module boundaries, and how many dependencies have already been loaded by other applications. The only way to determine whether native images benefit your application is by careful performance measurements in your key deployment scenarios. |

#### Improved Memory Use

Native images can significantly improve memory use when code is shared between processes. Native images are Windows PE files, so a single copy of a .dll file can be shared by multiple processes; by contrast, native code produced by the JIT compiler is stored in private memory and cannot be shared.

Applications that are run under terminal services can also benefit from shared code pages.

In addition, not loading the JIT compiler saves a fixed amount of memory for each application instance.

#### Faster Application Startup

Precompiling assemblies with Ngen.exe can improve the startup time for some applications. In general, gains can be made when applications share component assemblies because after the first application has been started the shared components are already loaded for subsequent applications. Cold startup, in which all the assemblies in an application must be loaded from the hard disk, does not benefit as much from native images because the hard disk access time predominates.

Hard binding can affect startup time, because all images that are hard bound to the main application assembly must be loaded at the same time.

|  |
| --- |
| **Description: NoteNote** |
| Before the .NET Framework version 3.5 Service Pack 1, you should put shared, strong-named components in the global assembly cache, because the loader performs extra validation on strong-named assemblies that are not in the global assembly cache, effectively eliminating any improvement in startup time gained by using native images. Optimizations that were introduced in the .NET Framework 3.5 SP1 removed the extra validation. |

#### Importance of Assembly Base Addresses

Because native images are Windows PE files, they are subject to the same rebasing issues as other executable files. The performance cost of relocation is even more pronounced if hard binding is employed.

To set the base address for a native image, use the appropriate option of your compiler to set the base address for the assembly. Ngen.exe uses this base address for the native image.

|  |
| --- |
| **Description: NoteNote** |
| Native images are larger than the managed assemblies from which they were created. Base addresses must be calculated to allow for these larger sizes. |

You can use a tool such as dumpbin.exe to view the preferred base address of a native image.

#### Summary of Usage Considerations

The following general considerations and application considerations may assist you in deciding whether to undertake the effort of evaluating native images for your application:

* Native images load faster than MSIL because they eliminate the need for many startup activities, such as JIT compilation and type-safety verification.
* Native images require a smaller initial working set because there is no need for the JIT compiler.
* Native images enable code sharing between processes.
* Native images require more hard disk space than MSIL assemblies and may require considerable time to generate.
* Native images must be maintained.
  + Images need to be regenerated when the original assembly or one of its dependencies is serviced.
  + A single assembly may need multiple native images for use in different applications or different scenarios. For example, the configuration information in two applications might result in different binding decisions for the same dependent assembly.
  + Native images must be generated by an administrator; that is, from a Windows account in the Administrators group.

In addition to these general considerations, the nature of your application must be considered when determining whether native images might provide a performance benefit:

* If your application runs in an environment that uses many shared components, native images allow the components to be shared by multiple processes.
* If your application uses multiple application domains, native images allow code pages to be shared across domains.

|  |
| --- |
| **Description: NoteNote** |
| In the .NET Framework versions 1.0 and 1.1, native images cannot be shared across application domains. This is not the case in version 2.0 or later. |

* If your application will be run under Terminal Server, native images allow sharing of code pages.
* Large applications generally benefit from compilation to native images. Small applications generally do not benefit.
* For long-running applications, run-time JIT compilation performs slightly better than native images. (Hard binding can mitigate this performance difference to some degree.)

### Hard Binding

Hard binding increases throughput and reduces working set size for native images. The disadvantage of hard binding is that all the images that are hard bound to an assembly must be loaded when the assembly is loaded. This can significantly increase startup time for a large application.

Hard binding is appropriate for dependencies that are loaded in all your application's performance-critical scenarios. As with any aspect of native image use, careful performance measurements are the only way to determine whether hard binding improves your application's performance.

The [DependencyAttribute](http://msdn.microsoft.com/en-us/library/system.runtime.compilerservices.dependencyattribute.aspx) and [DefaultDependencyAttribute](http://msdn.microsoft.com/en-us/library/system.runtime.compilerservices.defaultdependencyattribute.aspx) attributes allow you to provide hard binding hints to Ngen.exe.

|  |
| --- |
| **Description: NoteNote** |
| These attributes are hints to Ngen.exe, not commands. Using them does not guarantee hard binding. The meaning of these attributes may change in future releases. |

#### Specifying a Binding Hint for a Dependency

Apply the [DependencyAttribute](http://msdn.microsoft.com/en-us/library/system.runtime.compilerservices.dependencyattribute.aspx) to an assembly to indicate the likelihood that a specified dependency will be loaded. [LoadHint.Always](http://msdn.microsoft.com/en-us/library/zb5wbefd.aspx) indicates that hard binding is appropriate, [Default](http://msdn.microsoft.com/en-us/library/zb5wbefd.aspx) indicates that the default for the dependency should be used, and [Sometimes](http://msdn.microsoft.com/en-us/library/zb5wbefd.aspx) indicates that hard binding is not appropriate.

The following code shows the attributes for an assembly that has two dependencies. The first dependency (Assembly1) is an appropriate candidate for hard binding, and the second (Assembly2) is not.

Visual Basic

Imports System.Runtime.CompilerServices

<Assembly:DependencyAttribute("Assembly1", LoadHint.Always)>

<Assembly:DependencyAttribute("Assembly2", LoadHint.Sometimes)>

C#

using System.Runtime.CompilerServices;

[assembly:DependencyAttribute("Assembly1", LoadHint.Always)]

[assembly:DependencyAttribute("Assembly2", LoadHint.Sometimes)]

Visual C++

using namespace System::Runtime::CompilerServices;

[assembly:DependencyAttribute("Assembly1", LoadHint.Always)];

[assembly:DependencyAttribute("Assembly2", LoadHint.Sometimes)];

The assembly name does not include the file name extension. Display names can be used.

#### Specifying a Default Binding Hint for an Assembly

Default binding hints are only needed for assemblies that will be used immediately and frequently by any application that has a dependency on them. Apply the [DefaultDependencyAttribute](http://msdn.microsoft.com/en-us/library/system.runtime.compilerservices.defaultdependencyattribute.aspx) with [LoadHint.Always](http://msdn.microsoft.com/en-us/library/zb5wbefd.aspx) to such assemblies to specify that hard binding should be used.

|  |
| --- |
| **Description: NoteNote** |
| There is no reason to apply [DefaultDependencyAttribute](http://msdn.microsoft.com/en-us/library/system.runtime.compilerservices.defaultdependencyattribute.aspx) to .dll assemblies that do not fall into this category, because applying the attribute with any value other than [LoadHint.Always](http://msdn.microsoft.com/en-us/library/zb5wbefd.aspx) has the same effect as not applying the attribute at all. |

Microsoft uses the [DefaultDependencyAttribute](http://msdn.microsoft.com/en-us/library/system.runtime.compilerservices.defaultdependencyattribute.aspx) to specify that hard binding is the default for a very small number of assemblies in the .NET Framework, such as mscorlib.dll.

### Troubleshooting

To confirm that native images are being used by your application, you can use the [Fuslogvw.exe (Assembly Binding Log Viewer)](http://msdn.microsoft.com/en-us/library/e74a18c4.aspx). Select Native Images in the Log Categories box on the binding log viewer window. Fuslogvw.exe provides information about why a native image was rejected.

You can use the [jitCompilationStart MDA](http://msdn.microsoft.com/en-us/library/fw872k46.aspx) managed debugging assistant (MDA) to determine when the JIT compiler starts to compile a function.

### Deferred Processing

Generation of native images for a very large application can take considerable time. Similarly, changes to a shared component or changes to computer settings might require many native images to be updated. The install and update actions have a /queue option that queues the operation for deferred execution by the native image service. In addition, Ngen.exe has queue and executeQueuedItems actions that provide some control over the service. For more information, see [Native Image Service](http://msdn.microsoft.com/en-us/library/ms165074.aspx).

### Native Images and JIT Compilation

If Ngen.exe encounters any methods in an assembly that it cannot generate, it excludes them from the native image. When the runtime executes this assembly, it reverts to JIT compilation for the methods that were not included in the native image.

In addition, native images are not used if the assembly has been upgraded, or if the image has been invalidated for any reason.

#### Invalid Images

When you use Ngen.exe to create a native image of an assembly, the output depends upon the command-line options that you specify and certain settings on your computer. These settings include the following:

* The version of the .NET Framework.
* The version of the operating system, if the change is from the Windows 9x family to the Windows NT family.
* The exact identity of the assembly (recompilation changes identity).
* The exact identity of all assemblies that the assembly references (recompilation changes identity).
* Security factors.

Ngen.exe records this information when it generates a native image. When you execute an assembly, the runtime looks for the native image generated with options and settings that match the computer's current environment. The runtime reverts to JIT compilation of an assembly if it cannot find a matching native image. The following changes to a computer's settings and environment cause native images to become invalid:

* The version of the .NET Framework.

If you apply an update to the .NET Framework, all native images that you have created using Ngen.exe become invalid. For this reason, all updates of the .NET Framework execute the Ngen Update command, to ensure that all native images are regenerated. The .NET Framework automatically creates new native images for the .NET Framework libraries that it installs.

* The version of the operating system, if the change is from the Windows 9x family to the Windows NT family.

For example, if the version of the operating system running on a computer changes from Windows 98 to Windows XP, all native images stored in the native image cache become invalid. However, if the operating system changes from Windows 2000 to Windows XP, the images are not invalidated.

* The exact identity of the assembly.

If you recompile an assembly, the assembly's corresponding native image becomes invalid.

* The exact identity of any assemblies the assembly references.

If you update a managed assembly, all native images that directly or indirectly depend on that assembly become invalid and need to be regenerated. This includes both ordinary references and hard-bound dependencies. Whenever a software update is applied, the installation program should execute an Ngen Update command to ensure that all dependent native images are regenerated.

* Security factors.

Changing machine security policy to restrict permissions previously granted to an assembly can cause a previously compiled native image for that assembly to become invalid.

For detailed information about how the common language runtime administers code access security and how to use permissions, see [Code Access Security](http://msdn.microsoft.com/en-us/library/c5tk9z76.aspx).

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

The following command generates a native image for ClientApp.exe, located in the current directory, and installs the image in the native image cache. If a configuration file exists for the assembly, Ngen.exe uses it. In addition, native images are generated for any .dll files that ClientApp.exe references.

ngen install ClientApp.exe

An image installed with Ngen.exe is also called a root. A root can be an application or a shared component.

The following command generates a native image for MyAssembly.exe with the specified path.

ngen install c:\myfiles\MyAssembly.exe

When locating assemblies and their dependencies, Ngen.exe uses the same probing logic used by the common language runtime. By default, the directory that contains ClientApp.exe is used as the application base directory, and all assembly probing begins in this directory. You can override this behavior by using the /AppBase option.

|  |
| --- |
| **Description: NoteNote** |
| This is a change from Ngen.exe behavior in the .NET Framework versions 1.0 and 1.1, where the application base is set to the current directory. |

An assembly can have a dependency without a reference, for example if it loads a .dll file by using the [Assembly.Load](http://msdn.microsoft.com/en-us/library/system.reflection.assembly.load.aspx) method. You can create a native image for such a .dll file by using configuration information for the application assembly, with the /ExeConfig option. The following command generates a native image for MyLib.dll, using the configuration information from MyApp.exe.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl101_ctl00_ctl04_code');" \o "Copy Code)

ngen install c:\myfiles\MyLib.dll /ExeConfig:c:\myapps\MyApp.exe

Assemblies installed in this way are not removed when the application is removed.

To uninstall a dependency, use the same command-line options that were used to install it. The following command uninstalls the MyLib.dll from the previous example.

ngen uninstall c:\myfiles\MyLib.dll /ExeConfig:c:\myapps\MyApp.exe

To create a native image for an assembly in the global assembly cache, use the display name of the assembly. For example:

ngen install "ClientApp, Version=1.0.0.0, Culture=neutral,

PublicKeyToken=3c7ba247adcd2081, processorArchitecture=MSIL"

NGen.exe generates a separate set of images for each scenario you install. For example, the following commands install a complete set of native images for normal operation, another complete set for debugging, and a third for profiling:

ngen install MyApp.exe

ngen install MyApp.exe /debug

ngen install MyApp.exe /profile

### Displaying the Native Image Cache

Once native images are installed in the cache, they can be displayed using Ngen.exe. The following command displays all native images in the native image cache.

ngen display

The display action lists all the root assemblies first, followed by a list of all the native images on the computer.

Use the simple name of an assembly to display information only for that assembly. The following command displays all native images in the native image cache that match the partial name MyAssembly, their dependencies, and all roots that have a dependency on MyAssembly:

ngen display MyAssembly

Knowing what roots depend on a shared component assembly is useful in gauging the impact of an update action after the shared component is upgraded.

If you specify an assembly's file extension, you must either specify the path or execute Ngen.exe from the directory containing the assembly:

ngen display c:\myApps\MyAssembly.exe

The following command displays all native images in the native image cache with the name MyAssembly and the version 1.0.0.0.

ngen display "myAssembly, version=1.0.0.0"

### Updating Images

Images are typically updated after a shared component has been upgraded. To update all native images that have changed, or whose dependencies have changed, use the update action with no arguments.

ngen update

Updating all images can be a lengthy process. You can queue the updates for execution by the native image service by using the /queue option. For more information on the /queue option and installation priorities, see [Native Image Service](http://msdn.microsoft.com/en-us/library/ms165074.aspx).

ngen update /queue

### Uninstalling Images

Ngen.exe maintains a list of dependencies, so that shared components are removed only when all assemblies that depend on them have been removed. In addition, a shared component is not removed if it has been installed as a root.

The following command uninstalls all scenarios for the root ClientApp.exe:

ngen uninstall ClientApp

The uninstall action can be used to remove specific scenarios. The following command uninstalls all debug scenarios for ClientApp.exe:

ngen uninstall ClientApp /debug

|  |
| --- |
| **Description: NoteNote** |
| Uninstalling /debug scenarios does not uninstall a scenario that includes both /profile and /debug. |

The following command uninstalls all scenarios for a specific version of ClientApp.exe:

ngen uninstall "ClientApp, Version=1.0.0.0"

The following commands uninstall all scenarios for "ClientApp, Version=1.0.0.0, Culture=neutral, PublicKeyToken=3c7ba247adcd2081, processorArchitecture=MSIL", or just the debug scenario for that assembly:

ngen uninstall "ClientApp, Version=1.0.0.0, Culture=neutral,

PublicKeyToken=3c7ba247adcd2081, processorArchitecture=MSIL"

ngen uninstall "ClientApp, Version=1.0.0.0, Culture=neutral,

PublicKeyToken=3c7ba247adcd2081, processorArchitecture=MSIL" /debug

As with the install action, supplying an extension requires either executing Ngen.exe from the directory containing the assembly or specifying a full path.

For examples relating to the native image service, see [Native Image Service](http://msdn.microsoft.com/en-us/library/ms165074.aspx).

.NET Framework 4

**Managed Execution Process**

The managed execution process includes the following steps, which are discussed in detail later in this topic:

1. [Choosing a compiler](http://msdn.microsoft.com/en-us/library/k5532s8a.aspx#choosing_a_compiler).

To obtain the benefits provided by the common language runtime, you must use one or more language compilers that target the runtime.

1. [Compiling your code to MSIL](http://msdn.microsoft.com/en-us/library/k5532s8a.aspx#compiling_to_msil).

Compiling translates your source code into Microsoft intermediate language (MSIL) and generates the required metadata.

1. [Compiling MSIL to native code](http://msdn.microsoft.com/en-us/library/k5532s8a.aspx#compiling_msil_to_native_code).

At execution time, a just-in-time (JIT) compiler translates the MSIL into native code. During this compilation, code must pass a verification process that examines the MSIL and metadata to find out whether the code can be determined to be type safe.

1. [Running code](http://msdn.microsoft.com/en-us/library/k5532s8a.aspx#running_code).

The common language runtime provides the infrastructure that enables execution to take place and services that can be used during execution.

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifChoosing a Compiler

To obtain the benefits provided by the common language runtime (CLR), you must use one or more language compilers that target the runtime, such as Visual Basic, C#, Visual C++, F#, or one of many third-party compilers such as an Eiffel, Perl, or COBOL compiler.

Because it is a multilanguage execution environment, the runtime supports a wide variety of data types and language features. The language compiler you use determines which runtime features are available, and you design your code using those features. Your compiler, not the runtime, establishes the syntax your code must use. If your component must be completely usable by components written in other languages, your component's exported types must expose only language features that are included in the [Common Language Specification](http://msdn.microsoft.com/en-us/library/12a7a7h3.aspx) (CLS). You can use the [CLSCompliantAttribute](http://msdn.microsoft.com/en-us/library/system.clscompliantattribute.aspx) attribute to ensure that your code is CLS-compliant. For more information, see [Writing CLS-Compliant Code](http://msdn.microsoft.com/en-us/library/bhc3fa7f.aspx).

[Back to top](http://msdn.microsoft.com/en-us/library/k5532s8a.aspx#introduction)

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifCompiling to MSIL

When compiling to managed code, the compiler translates your source code into Microsoft intermediate language (MSIL), which is a CPU-independent set of instructions that can be efficiently converted to native code. MSIL includes instructions for loading, storing, initializing, and calling methods on objects, as well as instructions for arithmetic and logical operations, control flow, direct memory access, exception handling, and other operations. Before code can be run, MSIL must be converted to CPU-specific code, usually by a [just-in-time (JIT) compiler](http://msdn.microsoft.com/en-us/library/k5532s8a.aspx#compiling_msil_to_native_code). Because the common language runtime supplies one or more JIT compilers for each computer architecture it supports, the same set of MSIL can be JIT-compiled and run on any supported architecture.

When a compiler produces MSIL, it also produces metadata. Metadata describes the types in your code, including the definition of each type, the signatures of each type's members, the members that your code references, and other data that the runtime uses at execution time. The MSIL and metadata are contained in a portable executable (PE) file that is based on and that extends the published Microsoft PE and common object file format (COFF) used historically for executable content. This file format, which accommodates MSIL or native code as well as metadata, enables the operating system to recognize common language runtime images. The presence of metadata in the file together with MSIL enables your code to describe itself, which means that there is no need for type libraries or Interface Definition Language (IDL). The runtime locates and extracts the metadata from the file as needed during execution.

[Back to top](http://msdn.microsoft.com/en-us/library/k5532s8a.aspx#introduction)

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifCompiling MSIL to Native Code

Before you can run Microsoft intermediate language (MSIL), it must be compiled against the common language runtime to native code for the target machine architecture. The .NET Framework provides two ways to perform this conversion:

* A .NET Framework just-in-time (JIT) compiler.
* The .NET Framework [Ngen.exe (Native Image Generator)](http://msdn.microsoft.com/en-us/library/6t9t5wcf.aspx).

**Compilation by the JIT Compiler**

JIT compilation converts MSIL to native code on demand at application run time, when the contents of an assembly are loaded and executed. Because the common language runtime supplies a JIT compiler for each supported CPU architecture, developers can build a set of MSIL assemblies that can be JIT-compiled and run on different computers with different machine architectures. However, if your managed code calls platform-specific native APIs or a platform-specific class library, it will run only on that operating system.

JIT compilation takes into account the possibility that some code might never be called during execution. Instead of using time and memory to convert all the MSIL in a PE file to native code, it converts the MSIL as needed during execution and stores the resulting native code in memory so that it is accessible for subsequent calls in the context of that process. The loader creates and attaches a stub to each method in a type when the type is loaded and initialized. When a method is called for the first time, the stub passes control to the JIT compiler, which converts the MSIL for that method into native code and modifies the stub to point directly to the generated native code. Therefore, subsequent calls to the JIT-compiled method go directly to the native code.

**Install-Time Code Generation Using NGen.exe**

Because the JIT compiler converts an assembly's MSIL to native code when individual methods defined in that assembly are called, it affects performance adversely at run time. In most cases, that diminished performance is acceptable. More importantly, the code generated by the JIT compiler is bound to the process that triggered the compilation. It cannot be shared across multiple processes. To allow the generated code to be shared across multiple invocations of an application or across multiple processes that share a set of assemblies, the common language runtime supports an ahead-of-time compilation mode. This ahead-of-time compilation mode uses the [Ngen.exe (Native Image Generator)](http://msdn.microsoft.com/en-us/library/6t9t5wcf.aspx) to convert MSIL assemblies to native code much like the JIT compiler does. However, the operation of Ngen.exe differs from that of the JIT compiler in three ways:

* It performs the conversion from MSIL to native code before running the application instead of while the application is running.
* It compiles an entire assembly at a time, instead of one method at a time.
* It persists the generated code in the Native Image Cache as a file on disk.

**Code Verification**

As part of its compilation to native code, the MSIL code must pass a verification process unless an administrator has established a security policy that allows the code to bypass verification. Verification examines MSIL and metadata to find out whether the code is type safe, which means that it accesses only the memory locations it is authorized to access. Type safety helps isolate objects from each other and helps protect them from inadvertent or malicious corruption. It also provides assurance that security restrictions on code can be reliably enforced.

The runtime relies on the fact that the following statements are true for code that is verifiably type safe:

* A reference to a type is strictly compatible with the type being referenced.
* Only appropriately defined operations are invoked on an object.
* Identities are what they claim to be.

During the verification process, MSIL code is examined in an attempt to confirm that the code can access memory locations and call methods only through properly defined types. For example, code cannot allow an object's fields to be accessed in a manner that allows memory locations to be overrun. Additionally, verification inspects code to determine whether the MSIL has been correctly generated, because incorrect MSIL can lead to a violation of the type safety rules. The verification process passes a well-defined set of type-safe code, and it passes only code that is type safe. However, some type-safe code might not pass verification because of some limitations of the verification process, and some languages, by design, do not produce verifiably type-safe code. If type-safe code is required by the security policy but the code does not pass verification, an exception is thrown when the code is run.

[Back to top](http://msdn.microsoft.com/en-us/library/k5532s8a.aspx#introduction)

Description: http://i.msdn.microsoft.com/Global/Images/clear.gifRunning Code

The common language runtime provides the infrastructure that enables managed execution to take place and services that can be used during execution. Before a method can be run, it must be compiled to processor-specific code. Each method for which MSIL has been generated is JIT-compiled when it is called for the first time, and then run. The next time the method is run, the existing JIT-compiled native code is run. The process of JIT-compiling and then running the code is repeated until execution is complete.

During execution, managed code receives services such as garbage collection, security, interoperability with unmanaged code, cross-language debugging support, and enhanced deployment and versioning support.

In Microsoft Windows XP and Windows Vista, the operating system loader checks for managed modules by examining a bit in the COFF header. The bit being set denotes a managed module. If the loader detects managed modules, it loads mscoree.dll, and **\_CorValidateImage** and **\_CorImageUnloading** notify the loader when the managed module images are loaded and unloaded. **\_CorValidateImage** performs the following actions:

1. Ensures that the code is valid managed code.
2. Changes the entry point in the image to an entry point in the runtime.

On 64-bit Windows, **\_CorValidateImage** modifies the image that is in memory by transforming it from PE32 to PE32+ format.

.NET Framework 4

**Ilasm.exe (MSIL Assembler)**

Updated: April 2011

The MSIL Assembler generates a portable executable (PE) file from Microsoft intermediate language (MSIL). (For more information on MSIL, see [Managed Execution Process](http://msdn.microsoft.com/en-us/library/k5532s8a.aspx).) You can run the resulting executable, which contains MSIL and the required metadata, to determine whether the MSIL performs as expected.

This tool is automatically installed with Visual Studio and with the Windows SDK. To run the tool, we recommend that you use the Visual Studio Command Prompt or the Windows SDK Command Prompt (CMD Shell). These utilities enable you to run the tool easily, without navigating to the installation folder. For more information, see [Visual Studio and Windows SDK Command Prompts](http://msdn.microsoft.com/en-us/library/ms229859.aspx).

* If you have Visual Studio installed on your computer: On the taskbar, click **Start**, click **All Programs**, click **Visual Studio**, click **Visual Studio Tools**, and then click **Visual Studio Command Prompt**.

-or-

If you have the Windows SDK installed on your computer: On the taskbar, click **Start**, click **All Programs**, click the folder for the Windows SDK, and then click **Command Prompt** (or **CMD Shell**).

* At the command prompt, type the following:

ilasm [options] filename [[options]filename...]

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

|  |  |  |
| --- | --- | --- |
| **Argument** | **Description** | |
| *filename* | The name of the .il source file. This file consists of metadata declaration directives and symbolic MSIL instructions. Multiple source file arguments can be supplied to produce a single PE file with Ilasm.exe.  Description: Note**Note**  Ensure that the last line of code in the .il source file has either trailing white space or an end-of-line character. | |
| **Option** | | **Description** |
| **/alignment=***integer* | | Sets FileAlignment to the value specified by *integer* in the NT Optional header. If the .alignment IL directive is specified in the file, this option overrides it. |
| **/base=***integer* | | Sets ImageBase to the value specified by *integer* in the NT Optional header. If the .imagebase IL directive is specified in the file, this option overrides it. |
| **/clock** | | Measures and reports the following compilation times in milliseconds for the specified .il source file:  **Total Run**: The total time spent performing all the specific operations that follow.  **Startup**: Loading and opening the file.  **Emitting MD**: Emitting metadata.  **Ref to Def Resolution**: Resolving references to definitions in the file.  **CEE File Generation**: Generating the file image in memory.  **PE File Writing**: Writing the image to a PE file. |
| **/debug**[=**IMPL**|**OPT**] | | Includes debug information (local variable and argument names, and line numbers). Creates a PDB file.  **/debug** with no additional value disables JIT optimization and uses sequence points from the PDB file.  **IMPL** disables JIT optimization and uses implicit sequence points.  **OPT** enables JIT optimization and uses implicit sequence points.  **IMPL** and **OPT** are new in the .NET Framework version 2.0. |
| **/dll** | | Produces a .dll file as output. |
| **/enc**=*file* | | Creates Edit-and-Continue deltas from the specified source file.  This argument is for academic use only and is not supported for commercial use.  New in the .NET Framework version 2.0. |
| **/exe** | | Produces an executable file as output. This is the default. |
| **/flags=***integer* | | Sets ImageFlags to the value specified by *integer* in the common language runtime header. If the .corflags IL directive is specified in the file, this option overrides it. See CorHdr.h , COMIMAGE\_FLAGS for a list of valid values for *integer*. |
| **/fold** | | Folds identical method bodies into one.  New in the .NET Framework version 2.0. |
| **/include**=*includePath* | | Sets a path to search for files included with **#include**.  New in the .NET Framework version 2.0. |
| **/itanium** | | Specifies Intel Itanium as the target processor.  If no image bitness is specified, the default is **/pe64**.  New in the .NET Framework version 2.0. |
| **/key:***keyFile* | | Compiles *filename* with a strong signature using the private key contained in *keyFile*. |
| **/key:@***keySource* | | Compiles *filename* with a strong signature using the private key produced at *keySource*. |
| **/listing** | | Produces a listing file on the standard output. If you omit this option, no listing file is produced.  This parameter is not supported in .NET Framework version 2.0 or later. |
| **/mdv**=*versionString* | | Sets the metadata version string.  New in the .NET Framework version 2.0. |
| **/msv**=*major***.***minor* | | Sets the metadata stream version, where *major* and *minor* are integers.  New in the .NET Framework version 2.0. |
| **/noautoinherit** | | Disables default inheritance from [Object](http://msdn.microsoft.com/en-us/library/system.object.aspx) when no base class is specified.  New in the .NET Framework version 2.0. |
| **/nocorstub** | | Suppresses generation of the CORExeMain stub.  New in the .NET Framework version 2.0. |
| **/nologo** | | Suppresses the Microsoft startup banner display. |
| **/output:***file.ext* | | Specifies the output file name and extension. By default, the output file name is the same as the name of the first source file. The default extension is .exe. If you specify the **/dll** option, the default extension is .dll.  Description: Note**Note**  Specifying **/output:**myfile.dll does not set the **/dll** option. If you do not specify **/dll**, the result will be an executable file named myfile.dll. |
| **/optimize** | | Optimizes long instructions to short. For example, **br** to **br.s**.  New in the .NET Framework version 2.0. |
| **/pe64** | | Creates a 64-bit image (PE32+).  If no target processor is specified, the default is **/itanium**.  New in the .NET Framework version 2.0. |
| **/pdb** | | Creates a PDB file without enabling debug information tracking.  New in the .NET Framework version 2.0. |
| **/quiet** | | Specifies quiet mode; does not report assembly progress. |
| **/resource:***file.res* | | Includes the specified resource file in \*.res format in the resulting .exe or .dll file. Only one .res file can be specified with the **/resource** option. |
| **/stack**=*stackSize* | | Sets the SizeOfStackReserve value in the NT Optional header to *stackSize*.  New in the .NET Framework version 2.0. |
| **/stripreloc** | | Specifies that no base relocations are needed.  New in the .NET Framework version 2.0. |
| **/subsystem=***integer* | | Sets subsystem to the value specified by *integer* in the NT Optional header. If the .subsystem IL directive is specified in the file, this command overrides it. See winnt.h, IMAGE\_SUBSYSTEM for a list of valid values for *integer*. |
| **/x64** | | Specifies a 64-bit AMD processor as the target processor.  If no image bitness is specified, the default is **/pe64**.  New in the .NET Framework version 2.0. |
| **/?** | | Displays command syntax and options for the tool. |

|  |
| --- |
| **Description: NoteNote** |
| All options for Ilasm.exe are case-insensitive and recognized by the first three letters. For example, **/lis** is equivalent to **/listing** and **/res:**myresfile.res is equivalent to **/resource:**myresfile.res. Options that specify arguments accept either a colon (:) or an equal sign (=) as the separator between the option and the argument. For example, **/output:***file.ext* is equivalent to **/output=***file.ext*. |

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

The MSIL Assembler helps tool vendors design and implement MSIL generators. Using Ilasm.exe, tool and compiler developers can concentrate on MSIL and metadata generation without being concerned with emitting MSIL in the PE file format.

Similar to other compilers that target the runtime, such as C# and Visual Basic, Ilasm.exe does not produce intermediate object files and does not require a linking stage to form a PE file.

The MSIL Assembler can express all the existing metadata and MSIL features of the programming languages that target the runtime. This allows managed code written in any of these programming languages to be adequately expressed in MSIL Assembler and compiled with Ilasm.exe.

|  |
| --- |
| **Description: NoteNote** |
| Compilation might fail if the last line of code in the .il source file does not have either trailing white space or an end-of-line character. |

You can use Ilasm.exe in conjunction with its companion tool, [Ildasm.exe](http://msdn.microsoft.com/en-us/library/f7dy01k1.aspx). Ildasm.exe takes a PE file that contains MSIL code and creates a text file suitable as input to Ilasm.exe. This is useful, for example, when compiling code in a programming language that does not support all the runtime metadata attributes. After compiling the code and running the output through Ildasm.exe, the resulting MSIL text file can be hand-edited to add the missing attributes. You can then run this text file through the Ilasm.exe to produce a final executable file.

You can also use this technique to produce a single PE file from several PE files originally generated by different compilers.

|  |
| --- |
| **Description: NoteNote** |
| Currently, you cannot use this technique with PE files that contain embedded native code (for example, PE files produced by Visual C++). |

To make this combined use of Ildasm.exe and Ilasm.exe as accurate as possible, by default the assembler does not substitute short encodings for long ones you might have written in your MSIL sources (or that might be emitted by another compiler). Use the **/optimize** option to substitute short encodings wherever possible.

|  |
| --- |
| **Description: NoteNote** |
| Ildasm.exe only operates on files on disk. It does not operate on files installed in the global assembly cache. |

For more information about the grammar of MSIL, see the asmparse.grammar file in the Windows Software Development Kit (SDK).

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

The following command assembles the MSIL file myTestFile.il and produces the executable myTestFile.exe.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl35_ctl00_ctl00_code');" \o "Copy Code)

ilasm myTestFile

The following command assembles the MSIL file myTestFile.il and produces the .dll file myTestFile.dll.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl35_ctl00_ctl01_code');" \o "Copy Code)

ilasm myTestFile /dll

The following command assembles the MSIL file myTestFile.il and produces the .dll file myNewTestFile.dll.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl35_ctl00_ctl02_code');" \o "Copy Code)

ilasm myTestFile /dll /output:myNewTestFile.dll

The following code example shows an extremely simple application that displays "Hello World!" to the console. You can compile this code and then use the [Ildasm.exe](http://msdn.microsoft.com/en-us/library/f7dy01k1.aspx) tool to generate an MSIL file.

C#

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl35_ctl00_ctl04_code');" \o "Copy Code)

using System;

public class Hello

{

public static void Main(String[] args)

{

Console.WriteLine("Hello World!");

}

}

The following MSIL code example corresponds to the previous C# code example. You can compile this code into an assembly using the [Ilasm.exe (MSIL Assembler)](http://msdn.microsoft.com/en-us/library/496e4ekx.aspx) tool. Both MSIL and C# code examples display "Hello World!" to the console.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl35_ctl00_ctl06_code');" \o "Copy Code)

// Metadata version: v2.0.50215

.assembly extern mscorlib

{

.publickeytoken = (B7 7A 5C 56 19 34 E0 89 ) // .z\V.4..

.ver 2:0:0:0

}

.assembly sample

{

.custom instance void [mscorlib]System.Runtime.CompilerServices.CompilationRelaxationsAttribute::.ctor(int32) = ( 01 00 08 00 00 00 00 00 )

.hash algorithm 0x00008004

.ver 0:0:0:0

}

.module sample.exe

// MVID: {A224F460-A049-4A03-9E71-80A36DBBBCD3}

.imagebase 0x00400000

.file alignment 0x00000200

.stackreserve 0x00100000

.subsystem 0x0003 // WINDOWS\_CUI

.corflags 0x00000001 // ILONLY

// Image base: 0x02F20000

// =============== CLASS MEMBERS DECLARATION ===================

.class public auto ansi beforefieldinit Hello

extends [mscorlib]System.Object

{

.method public hidebysig static void Main(string[] args) cil managed

{

.entrypoint

// Code size 13 (0xd)

.maxstack 8

IL\_0000: nop

IL\_0001: ldstr "Hello World!"

IL\_0006: call void [mscorlib]System.Console::WriteLine(string)

IL\_000b: nop

IL\_000c: ret

} // end of method Hello::Main

.method public hidebysig specialname rtspecialname

instance void .ctor() cil managed

{

// Code size 7 (0x7)

.maxstack 8

IL\_0000: ldarg.0

IL\_0001: call instance void [mscorlib]System.Object::.ctor()

IL\_0006: ret

} // end of method Hello::.ctor

} // end of class Hello

.NET Framework 4

**Ildasm.exe (MSIL Disassembler)**

Updated: April 2011

The MSIL Disassembler is a companion tool to the MSIL Assembler (Ilasm.exe). Ildasm.exe takes a portable executable (PE) file that contains Microsoft intermediate language (MSIL) code and creates a text file suitable as input to Ilasm.exe.

This tool is automatically installed with Visual Studio and with the Windows SDK. To run the tool, we recommend that you use the Visual Studio Command Prompt or the Windows SDK Command Prompt (CMD Shell). These utilities enable you to run the tool easily, without navigating to the installation folder. For more information, see [Visual Studio and Windows SDK Command Prompts](http://msdn.microsoft.com/en-us/library/ms229859.aspx).

* If you have Visual Studio installed on your computer: On the taskbar, click **Start**, click **All Programs**, click **Visual Studio**, click **Visual Studio Tools**, and then click **Visual Studio Command Prompt**.

-or-

If you have the Windows SDK installed on your computer: On the taskbar, click **Start**, click **All Programs**, click the folder for the Windows SDK, and then click **Command Prompt** (or **CMD Shell**).

* At the command prompt, type the following:

ildasm [options] [PEfilename] [options]

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

The following options are available for .exe, .dll, .obj, and .lib files.

|  |  |
| --- | --- |
| **Option** | **Description** |
| **/out=***filename* | Creates an output file with the specified *filename*, rather than displaying the results in a graphical user interface. |
| **/rtf** | Produces output in rich text format. Invalid with the **/text** option.  New in the .NET Framework version 2.0. |
| **/text** | Displays the results to the console window, rather than in a graphical user interface or as an output file. |
| **/html** | Produces output in HTML format. Valid with the **/output** option only.  New in the .NET Framework version 2.0. |
| **/?** | Displays the command syntax and options for the tool. |

The following additional options are available for .exe and .dll files.

|  |  |
| --- | --- |
| **Option** | **Description** |
| **/bytes** | Shows actual bytes, in hexadecimal format, as instruction comments. |
| **/caverbal** | Produces custom attribute blobs in verbal form. The default is binary form.  New in the .NET Framework version 2.0. |
| **/linenum** | Includes references to original source lines. |
| **/nobar** | Suppresses the disassembly progress indicator pop-up window. |
| **/noca** | Suppresses the output of custom attributes.  New in the .NET Framework version 2.0. |
| **/pubonly** | Disassembles only public types and members. Equivalent to **/visibility:PUB**. |
| **/quoteallnames** | Includes all names in single quotes. |
| **/raweh** | Shows exception handling clauses in raw form. |
| **/source** | Shows original source lines as comments. |
| **/tokens** | Shows metadata tokens of classes and members. |
| **/visibility:***vis*[+*vis*...] | Disassembles only types or members with the specified visibility. The following are valid values for *vis*:  **PUB** — Public  **PRI** — Private  **FAM** — Family  **ASM** — Assembly  **FAA** — Family and Assembly  **FOA** — Family or Assembly  **PSC** — Private Scope  For definitions of these visibility modifiers, see [MethodAttributes](http://msdn.microsoft.com/en-us/library/system.reflection.methodattributes.aspx) and [TypeAttributes](http://msdn.microsoft.com/en-us/library/system.reflection.typeattributes.aspx). |

The following options are valid for .exe and .dll files for file or console output only.

|  |  |
| --- | --- |
| **Option** | **Description** |
| **/all** | Specifies a combination of the **/header**, **/bytes**, **/stats**, **/classlist**, and **/tokens** options.  Description: Note**Note**  In the .NET Framework versions 1.0 and 1.1, specifies a combination of the **/header**, **/bytes**, and **/tokens** options. |
| **/classlist** | Includes a list of classes defined in the module.  New in the .NET Framework version 2.0. |
| **/forward** | Uses forward class declaration.  New in the .NET Framework version 2.0. |
| **/headers** | Includes file header information in the output. |
| **/item:***class*[**::***member*[**(***sig*]] | Disassembles the following depending upon the argument supplied:   * Disassembles the specified *class*. * Disassembles the specified *member* of the *class*. * Disassembles the *member*of the *class* with the specified signature *sig*. The format of *sig* is:   [**instance**] *returnType*(*parameterType1*, *parameterType2*, …, *parameterTypeN*)  **Note** In the .NET Framework versions 1.0 and 1.1, *sig* must be followed by a closing parenthesis: (*sig*). In version 2.0 the closing parenthesis must be omitted: (*sig*. |
| **/noil** | Suppresses MSIL assembly code output. |
| **/stats** | Includes statistics on the image.  New in the .NET Framework version 2.0. |
| **/typelist** | Produces the full list of types, to preserve type ordering in a round trip.  New in the .NET Framework version 2.0. |
| **/unicode** | Uses Unicode encoding for the output. |
| **/utf8** | Uses UTF-8 encoding for the output. ANSI is the default. |

The following options are valid for .exe, .dll, .obj, and .lib files for file or console output only.

|  |  |
| --- | --- |
| **Option** | **Description** |
| **/metadata**[=*specifier*] | Shows metadata, where *specifier* is:  **MDHEADER** — Show the metadata header information and sizes.  **HEX** — Show information in hex as well as in words.  **CSV** — Show the record counts and heap sizes.  **UNREX** — Show unresolved externals.  **SCHEMA** — Show the metadata header and schema information.  **RAW** — Show the raw metadata tables.  **HEAPS** — Show the raw heaps.  **VALIDATE** — Validate the consistency of the metadata.  You can specify **/metadata** multiple times, with different values for *specifier*.  New in the .NET Framework version 2.0. |

The following options are valid for .lib files for file or console output only.

|  |  |
| --- | --- |
| **Option** | **Description** |
| **/objectfile**=*filename* | Shows the metadata of a single object file in the specified library.  New in the .NET Framework version 2.0. |
| **Description: NoteNote** | |
| All options for Ildasm.exe are case-insensitive and recognized by the first three letters. For example, **/quo** is equivalent to **/quoteallnames**. Options that specify arguments accept either a colon (:) or an equal sign (=) as the separator between the option and the argument. For example, **/output:***filename* is equivalent to **/output=***filename*. | |

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

Ildasm.exe only operates on PE files on disk. It does not operate on files installed in the global assembly cache.

The text file produced by Ildasm.exe can be used as input to the MSIL Assembler (Ilasm.exe). This is useful, for example, when compiling code in a programming language that does not support all the runtime metadata attributes. After compiling the code and running its output through Ildasm.exe, the resulting MSIL text file can be hand-edited to add the missing attributes. You can then run this text file through the MSIL Assembler to produce a final executable file.

|  |
| --- |
| **Description: NoteNote** |
| Currently, you cannot use this technique with PE files that contain embedded native code (for example, PE files produced by Visual C++). |

You can use the default GUI in the MSIL Disassembler to view the metadata and disassembled code of any existing PE file in a hierarchical tree view. To use the GUI, type **ildasm** at the command line without supplying the *PEfilename* argument or any options. From the **File** menu, you can navigate to the PE file that you want to load into Ildasm.exe. To save the metadata and disassembled code displayed for the selected PE, select the **Dump** command from the **File** menu. To save the hierarchical tree view only, select the **Dump Treeview** command from the **File** menu. For a detailed guide to loading a file into Ildasm.exe and interpreting the output, see the Ildasm.exe Tutorial, located in the Samples folder that ships with the Windows Software Development Kit (SDK).

If you provide Ildasm.exe with a *PEfilename* argument that contains embedded resources, the tool produces multiple output files: a text file that contains MSIL code and, for each embedded managed resource, a .resources file produced using the resource's name from metadata. If an unmanaged resource is embedded in *PEfilename*, a .res file is produced using the filename specified for MSIL output by the **/output** option*.*

|  |
| --- |
| **Description: NoteNote** |
| Ildasm.exe shows only metadata descriptions for .obj and .lib input files. MSIL code for these file types is not disassembled. |

You can run Ildasm.exe over an.exe or .dll file to determine whether the file is managed. If the file is not managed, the tool displays a message stating that the file has no valid common language runtime header and cannot be disassembled. If the file is managed, the tool runs successfully.

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

The following command causes the metadata and disassembled code for the PE file MyHello.exe to display in the Ildasm.exe default GUI.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl33_ctl00_ctl00_code');" \o "Copy Code)

ildasm myHello.exe

The following command disassembles the file MyFile.exe and stores the resulting MSIL Assembler text in the file MyFile.il.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl33_ctl00_ctl01_code');" \o "Copy Code)

ildasm MyFile.exe /output:MyFile.il

The following command disassembles the file MyFile.exe and displays the resulting MSIL Assembler text to the console window.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl33_ctl00_ctl02_code');" \o "Copy Code)

ildasm MyFile.exe /text

If the file MyApp.exe contains embedded managed and unmanaged resources, the following command produces four files: MyApp.il, MyApp.res, Icons.resources, and Message.resources:

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl33_ctl00_ctl03_code');" \o "Copy Code)

ildasm MyApp.exe /output:MyApp.il

The following command disassembles the method MyMethod within the class MyClass in MyFile.exe and displays the output to the console window.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl33_ctl00_ctl04_code');" \o "Copy Code)

ildasm /item:MyClass::MyMethod MyFile.exe /text

In the previous example, there could be several methods named MyMethod with different signatures. The following command disassembles the instance method MyMethod with the return type of **void** and the parameter types **int32** and **string**.

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl33_ctl00_ctl05_code');" \o "Copy Code)

ildasm /item:"MyClass::MyMethod(instance void(int32,string)" MyFile.exe /text

|  |
| --- |
| **Description: NoteNote** |
| In the .NET Framework versions 1.0 and 1.1, the left parenthesis that follows the method name must be balanced by a right parenthesis after the signature: MyMethod(instance void(int32)). In the .NET Framework version 2.0 the closing parenthesis must be omitted: MyMethod(instance void(int32). |

To retrieve a **static** method (**Shared** method in Visual Basic), omit the keyword **instance**. Class types that are not primitive types like **int32** and **string** must include the namespace and must be preceded by the keyword **class**. External types must be preceded by the library name in square brackets. The following command disassembles a static method named MyMethod that has one parameter of type [AppDomain](http://msdn.microsoft.com/en-us/library/system.appdomain.aspx) and has a return type of [AppDomain](http://msdn.microsoft.com/en-us/library/system.appdomain.aspx).

[Copy Code](javascript:CopyCode('ctl00_MTCS_main_ctl33_ctl00_ctl09_code');" \o "Copy Code)

ildasm /item:"MyClass::MyMethod(class [mscorlib]System.AppDomain(class [mscorlib]System.AppDomain)" MyFile.exe /text

A nested type must be preceded by its containing class, delimited by a forward slash. For example, if the MyNamespace.MyClass class contains a nested class named NestedClass, the nested class is identified as follows: class MyNamespace.MyClass/NestedClass.