

Command-Line Tools

This section reviews the syntax of the command-line utilities that come with the .NET Framework and that are related to creating and managing assemblies. In most situations, you can perform many of these tasks from inside Visual Studio's dialog boxes or by adding the proper attribute to your source code. However, several jobs require you to run command-line tools because they offer more options or because you might need to automate a task by means of a script or a batch file.

The simplest way to run all the command-line tools mentioned in this section is to open a command window using the Visual Studio .NET Command Prompt command on the Start menu. Or you can open a regular prompt window and then launch the corvars.bat program to correctly initialize the PATH environment variable so that you don't have to specify the complete path of these commands. (You can find the corvars.bat program in the C:\Program Files\Microsoft Visual Studio .NET\FrameworkSDK\Bin directory.) For example, I keep a copy of corvars.bat in the C:\ directory so that I can run it easily from both the command prompt and a batch program.

The Visual Basic Compiler (VBC)

VBC.exe is more than a plain compiler, in that by default it creates single-file assemblies without the aid of a linker. It can compile one or more source code files, and it produces an executable that's a self-sufficient assembly containing its own manifest. By specifying the /t:module option, you create a single module that you later add to a multifile assembly by using the AL utility, as I explain in the next section.

The simplest operation that you can perform with the VBC command-line compiler is the production of a Windows console application. To see how to proceed, create the following Module.vb and Teller.vb files (using Notepad or from inside the IDE):

```
' The Module.vb module
Module MainModule
    Sub Main
        Dim t As New Teller
        t.Say("Hello world")
    End Sub
End Module

' The Teller.vb file
Imports System

Class Teller
    Sub Say(ByVal msg As String)
        Console.WriteLine(msg)
    End Sub
End Class
```

Compiling these modules is easy:

```
vbc module.vb teller.vb
```

The preceding statement produces the executable module.exe file that, when run, displays the "Hello world" string in the console window. The name of the executable

file is derived from the name of the first source file in the list, but you can specify another name by using the /out option. This option is especially useful when you use wildcards to compile all the source files in a directory, as in this line of code:

```
vbc /out:myprog *.vb
```

You can use the /recurse option to compile all the source files in the current directory and all its directories:

```
vbc /out:myprog /recurse:*.vb
```

Another useful option is /imports, which lets you specify projectwide Imports. (You can import multiple namespaces by separating each with a comma.) For example, you might delete the Imports statement in Teller.vb and be able to recompile the project anyway using this command:

```
vbc /out:myprog /imports:system *.vb
```

The /main option is necessary when two or more classes contain a Sub Main procedure so that the compiler can understand which procedure is the entry point for the application:

```
vbc /out:myprog /main:startclass *.vb
```

All the applications created so far were console applications. You can create other types of applications using the /target option, which can be shortened to /t. For example, you can create a standard Windows application with this code:

```
vbc /t:winexe /out:myprog *.vb
```

and a class library in a DLL with

```
vbc /t:library /out:mylib *.vb
```

Table 1 summarizes all the options that the VBC compiler supports. The rightmost column describes the corresponding option offered by Visual Studio .NET when applicable. In most cases, these options can be found in the project Property Pages dialog box. The most important option that can't be activated from inside Visual Studio .NET is /t:module, which creates a module that you can later add to an assembly by using the AL.exe utility:

```
vbc /t:module /out:mymodule *.vb
```

Note that if the program being compiled uses types defined in another module that was compiled using /target:module, you must use the / addmodule option to specify the referenced module:

```
vbc myapp /addmodule:other.netmodule
```

The other module becomes part of the application's assembly, and both executable files must be in the same directory when the application runs.

Table 1 Available Command-Line Options for the VBC Compiler and the Corresponding Options Inside Visual Basic .NET

Category	Option	Description	Visual Studio .NET Command
Output file	/out:outfile	Specifies the output filename.	Assembly Name (General page) and Output Path (Configuration Properties, Build page)
	/t[arget]:exe	Creates a console application (default if omitted).	Output Type (General page)
	/t[arget]:winexe	Creates a Windows application.	Output Type (General page)
	/t[arget]:library	Creates a library assembly.	Output Type (General page)
	/t[arget]:module	Creates a .netmodule module that can be added to an assembly.	Not available
	/main:class	Specifies the class that contains Sub Main.	Startup Object (General page)
	/baseaddress:addr	Specifies the default base address for the DLL, in hexadecimal.	DLL Base Address (Configuration Properties, Optimizations page)
Source files	filespec ...	Specifies one or more source files to compile; the file specification can include wildcards.	The list of files loaded in the project (as shown in Solution Explorer)
	/recurse:filespec	Searches subdirectories for all files to compile; the file specification can include wildcards.	Not available
References	/r[eference]:assembly	References an external assembly.	The Add Reference dialog box
	/libpath:dir	Specifies the path for external assemblies referenced with /r.	Not available
	/ imports:namespace,...	Imports a namespace from one or more referenced assemblies.	Project Imports (Imports page)

Category	Option	Description	Visual Studio .NET Command
	/addmodule:filename	Makes all the type information from a file compiled with /t:module available to the program being compiled.	Not available
Language settings	/rootnamespace:name	Defines the root namespace for this project.	Root Namespace (General page)
	/removeintchecks[+/-]	Removes integer checks.	Remove Integer Overflow Checks (Configuration Properties, Optimizations page)
	/d[efine]:name=value,...	Defines one or more compilation constants.	Custom Constants (Configuration Properties, Build page)
	/debug[+/-]	Generates debug information.	Generate Debugging Information (Configuration Properties, Build page)
	/nowarn	Suppresses the compiler's ability to generate warnings.	Enable Build Warnings (Configuration Properties, Build page)
	/warnaserror	Promotes warnings to errors.	Treat Compiler Warnings As Errors (Configuration Properties, Build page)
	/optionexplicit[+/-]	Requires explicit variable declarations.	Option Explicit (Common Properties, Build page)
	/optionstrict[+/-]	Enforces strict language semantics.	Option Strict (Common Properties, Build page)
	/optioncompare:text	Specifies text-style (case-insensitive) string comparisons.	Option Compare (Common Properties, Build page)
	/optioncompare:binary	Specifies binary-style (case sensitive) string comparisons.	Option Compare (Common Properties, Build page)
	/optimize[+/-]	Enables optimizations.	Enable Optimizations (Configuration Properties, Optimizations page)

Category	Option	Description	Visual Studio .NET Command
Resources	/res[ource]:filename	Adds a file internally to the assembly. See SDK documentation for additional information on complete syntax.	Loaded resource files (Solution Explorer)
	/ linkres[ource]:filename	Links a file externally to the assembly. See SDK documentation for additional information on complete syntax.	Not available
	/win32icon:iconfile	Specifies an icon .ico file.	Application Icon (Common Properties, Build page)
	/ win32resource:resfile	Specifies a Win32 resource file.	Not available
Assembly settings	/keyfile:filename	Specifies key file for the assembly.	Not available
	/keycontainer:string	Specifies key container name for the assembly.	Not available
	/version:ma.mi.re.bu	Specifies a version in the format major.minor.revision.build.	Not available
Miscellaneous	/nologo	Suppresses the display of the compiler copyright banner.	n/a
	/? or /help	Displays this list of available options.	n/a
	@responsefile	Retrieves command- line settings from a response file.	n/a
	/bugreport:filename	Creates a bug report file.	n/a
	/utf8output	Displays compiler output using UTF-8 encoding.	n/a
	/quiet	Suppresses display of source code lines for syntax errors.	n/a

The Assembly Linker (AL)

The Assembly Linker can take one or more managed modules (and zero or more resource files) and combine them to deliver an assembly—or more precisely, another module that contains an assembly manifest. For example, the following commands produce an assembly from the Teller.vb module:

```
vbc /t:module teller.vb
al teller.mcm /t:lib /out:mylib.dll /version:1.2.3.4
```

The /version switch supports several variants, other than the canonical major.minor.revision.build format. You can

- Specify only the major number—the minor, revision, and build numbers will be 0.
- Specify only the major and minor numbers—the revision and build numbers will be 0.
- Specify the major and minor numbers followed by an asterisk (1.2.*)—the revision number will be set equal to the number of days since January 1, 2000, and the build number will be set equal to the number of seconds since midnight, divided by 2.
- Specify the major, minor, and revision numbers followed by an asterisk (1.2.3.*)—the build number will be set equal to the number of seconds since midnight, divided by 2.

For example, I ran the following command around noon on March 31, 2000:

```
al teller.mcm /t:lib /out:mylib.dll /version:1.2.* /comp:VB2THEMAX
/trade:"(C) 2000 Francesco Balena"
```

Then I right-clicked the teller.dll file from Windows Explorer and verified that the file version was 1.2.424.18558. (See Figure 1.) Also, note that you can set many other version-related properties, such as the company name and the legal trademark strings, by means of appropriate switches on the command line. See Table 2 for a complete list of supported switches.

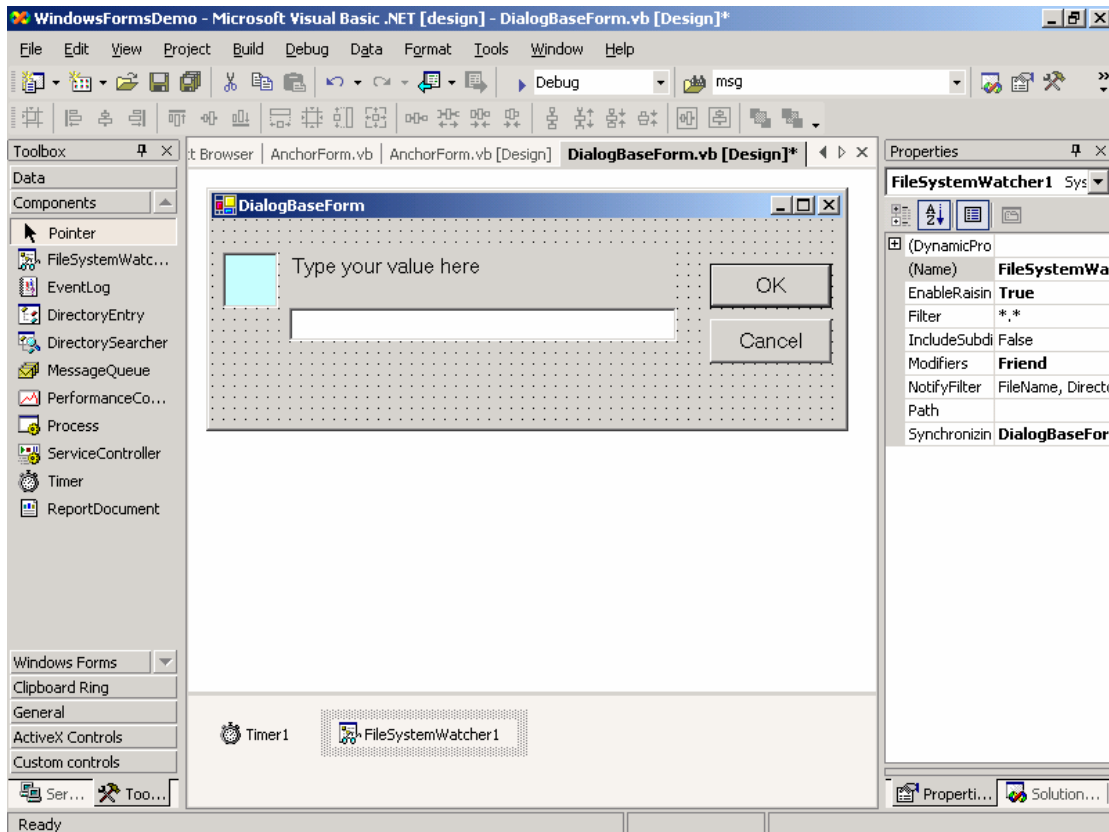


Figure 1

The Version tab of the Properties window for the mylib.dll file shows the revision and build generated by the AL.

I've already discussed a few other options in the "Partial Signing and Key Containers" section, such as /delaysign for partial (delayed) signing and /keyfile and /keyname for signing the assembly with a public key.

Table 2 Available Command-Line Options for the AL Assembly Linker Tool

Category	AL Option	Description
Source	filename [,targetfile]	A managed module that doesn't contain a manifest; if targetfile is provided, AL copies the file there and then begins the compilation.
	/embed[resource]: file	A resource file to be embedded into the PE file being created.
	/link[resource]: file	A resource file to be linked from the PE file being created.
Output file	/out:filename	The name of the file that will contain the assembly manifest.
	/t[arget]:lib exe win	The type of the output file: a DLL library, a console application, a Windows GUI application.

Category	AL Option	Description
	/main: class.method	The method that works as the entry point to the assembly.
	/base[address]: hexaddr	The base address where the DLL will be loaded (hexadecimal).
Resources	/win32res:file	Embeds a Win32 resource file (.res) in the output file.
	/win32icon:file	Inserts a Win32 icon file (.ico) in the output file.
Version	/version: ma.mi.rev.build	The assembly version number. You can omit any version portion to the right of major; you can use an asterisk for rev.build or build, in which case rev will be the number of days since 1/1/2000 and build the number of seconds since midnight divided by 2.
	/fileversion:version	A value for the File Version string in the assembly.
	/c[culture]: culture	The supported culture.
	/productv[ersion]: "text"	Informational version: a string version to be used in product and marketing literature. You need to enclose the string in double quotation marks if it contains spaces and other punctuation characters. (This rule applies to all of the following options in the Version category.)
	/product:"text"	The name of the product.
	/comp[any]:"text"	The company name.
	/copy[right]:"text"	The copyright string.
	/trade[mark]:"text"	The trademark string.
	/descr[iption]:"text"	The product description string.
	/config[uration]:"text"	The configuration string.
	/title:"text"	The title for the assembly.
	/template:filename	Specifies another assembly from which to inherit all the metadata except the culture field; this is especially useful when building satellite assemblies.
Assembly settings	/algid:id	The algorithm to use to hash files in a multifile assembly; id is a hexadecimal value equal to CALG_SHA1 (default) or CALG_MD5.

Category	AL Option	Description
	/delay[sign][+/-]	Marks the assembly for partial or delayed signing (requires /keyfile or /keyname).
	/keyf[ile]:file	Specifies a file that contains the public/ private key to make a shared assembly (or only the public key if partial signing).
	/keyn[ame]:file	Specifies a key container for the public/ private key pair.
	/e[vidence]:file	Embeds a file in the assembly with the resource name Security.Evidence.
	/flags:hexvalue	A value for the Flag field in the assembly; hexvalue can be 0 (side-by-side compatible), 10 (incompatible with other versions in the same AppDomain), 20 (incompatible with other versions in the same process), or 30 (incompatible with other versions in the same system).
Miscellaneous	/nologo	Suppresses the display of the compiler copyright banner.
	/? or /help	Displays this list of available options.
	@responsefile	Retrieves command-line settings from a response file.
	/bugreport:filename	Creates a bug report file.
	/fullpaths	Displays complete filenames in error messages.

The Strong Name Utility (SN)

The SN utility helps you create assemblies with strong names and is therefore necessary to create shared assemblies. It has several options, only one of which can be specified at a given time.

The most useful command is `-k`, which creates a random public/ private key pair and stores it in a file of your choice:

```
sn -k mykey.snk
```

This command produces the `mykey.snk` file, which contains the key pair.

The `-p` command extracts the public key from an `.snk` file and stores it in another file, ready to be referenced by the `/keyfile` option of the VBC compiler or AL linker when you're doing delayed signing:

```
sn -p mykey.snk public.key
```

The file produced by the `-p` command contains the public key in a binary, nonreadable format. You can use the `-o` command to get it in a textual CSV (comma-separated value) format, which can be useful for initializing an array in source code:

```
sn -o mykey.snk public.csv
```

(The result is copied into the Clipboard if you omit the filename.) You can also extract the public key from an existing assembly by using the `-e` command:

```
sn -e mylib.dll public.key
```

The `-T` and `-t` commands display the token of the public key contained in an assembly file or an `.snk` file, respectively. As I explained in the “Private and Shared Assemblies” section, the token is an 8-byte hash value of the public key; the runtime uses this token instead of the full public key when referencing the assembly from another assembly. You can use `-Tp` or `-tp` to display the public key together with the token:

```
sn -tp mykey.snk
```

The `-v` option simply verifies the shared name in a given assembly—that is, it checks that the hash value is correct:

```
sn -v mylib.dll
```

The `-Vr` option registers the assembly for verification skipping (so that you can install it in the GAC without an actual private key), and the `-R` command re-signs a previously signed or a partially signed assembly using the key pair in the specified file:

```
sn -R mylib.dll mykey.snk
```

For additional information about the SN utility, see the .NET Framework SDK documentation.

The Global Assembly Cache Utility (GACUTIL)

The GACUTIL tool lets you manage the global assembly cache from the command line, scripts, or make files. It provides the same functionality as the `shfusion.dll` GAC viewer, which you transparently use when you're browsing the GAC with Windows Explorer.

The GACUTIL tool supports only a handful of commands, which I summarize in Table 3. For example, the following command installs the `mylib.dll` assembly in the global assembly cache:

```
gacutil /i mylib.dll
```

You can remove this assembly from the GAC with the following command:

```
gacutil /u mylib
```

(Note that you pass the assembly name, not the filename.) A potential problem with the /u command is that it removes every instance of the mylib assembly, regardless of its version number, culture, and public key. To avoid undesirable deletions, you should specify precisely what you want to delete, as follows:

```
gacutil -u mylib,ver=1.2.00,loc=it,PublicKeyToken=44cd4ab12f0a88a1
```

Finally, you can list all the shared assemblies in the GAC using this command:

```
gacutil /l
```

Table 3 Available Command-Line Options for the GACUTIL Tool

Option	Description
/i filename	Installs an assembly in the GAC.
/if filename	Installs an assembly in the GAC and forces an overwrite if the assembly exists already.
/ir filename refscheme id descr	Installs an assembly in the GAC with a traced reference; the argument must include the filename, the reference scheme (typically FILEPATH), the ID (typically the path of the assembly), and a description. Example: gacutil /ir myasm.dll FILEPATH c:\myprog\myasm.dll MyAssembly.
/u asmname	Removes an assembly from the GAC. The name of the assembly can include version, public key token, and locale (MyAsm,Version=1.2.0.0,Culture=en,PublicKeyToken= 874e23ab874e23ab).
/ur asmname refscheme id descr	Removes an assembly reference from the GAC. The name of the assembly can include version, public key token, and locale; other arguments are the same as for /ir.
/uf asmname	Removes an assembly reference from the GAC. The name of the assembly can include version, public key token, and locale. The assembly will be removed only if not referenced by Windows Installer.
/ungen asmname	Removes an assembly from the cache of pre-JITted files.
/l	Lists all the assemblies in the GAC and the assemblies in the cache of pre-JITted files.
/lr	Lists all the assemblies in the GAC with traced reference information.
/ldl	Lists the contents of the downloaded files cache.
/cdl	Deletes the contents of the download cache.
/nologo	Suppresses display of logo.
/silent	Suppresses display of all output.
/? or /help	Displays a help page.

The MSIL Disassembler (ILDASM)

I cover the ILDASM tool in the “Microsoft Intermediate Language (MSIL)” section in Chapter 13. I’ll just focus here on the options you can specify from the command line.

A great feature of the ILDASM tool is its ability to generate text files that can be fed to the MSIL Assembler tool (ILASM), which is a compiler that accepts MSIL code directly. This feature enables you to perform a *code round-trip*—that is, first create an executable using a more traditional, higher-level, language and then produce the MSIL code, edit it (for example, to add attributes not supported by the high-level language), and pass the edited source code to the ILASM tool. You’ve seen an example of round-trip code in Chapter 1.

ILDASM supports a group of advanced command-line settings, which you must explicitly enable using the /adv switch. This option works also when you’re working with a graphical user interface and extends the View menu with additional commands. (See Figure 2.)

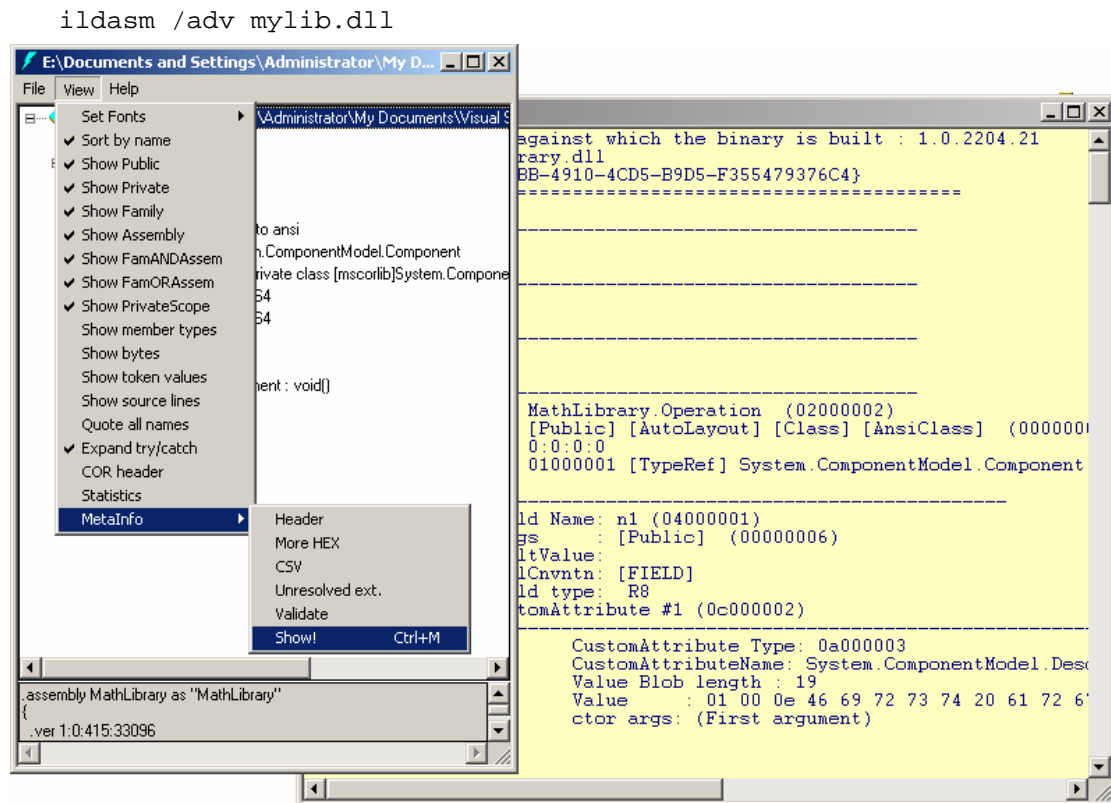


Figure 2

The menu commands added by the /adv option. The background window shows a sample of the output produced by clicking Show! on the MetaInfo submenu.

The following command outputs complete metadata information in a file to the console window:

```
ildasm /adv /text /all mylib.dll
```

See Table 4 for the complete list of all ILDASM command-line options. Note that all options are case insensitive and can be shortened to three characters (/BYT instead of /bytes). You can use the = separator instead of the colon character.

Table 4 Available Command-Line Options for the ILDASM Tool

Category	Option	Description
Output destination	/output:filename	Sends output to a file instead of a window.
	/text	Sends output to a console window.
Scope	/visibility:vis [+vis...]	Disassembles only types with specified visibility; vis can be PUB, PRI, FAM, ASM, FAA, FOA, or PSC.
	/pubonly	Disassembles only public types (same as /visibility:PUB).
Output contents	/all	Combines the /header, /bytes, and /tokens options.
	/bytes	Shows actual bytes as instruction comments (in hex).
	/raweh	Shows exception handler clauses in raw form.
	/tokens	Shows metadata tokens of classes and members.
	/source	Shows original source lines as comments.
	/linenum	Includes references to original source lines.
	/quoteallnames	Includes all names in single quotation marks.
Options for file/console output only	/utf8	Uses UTF-8 encoding for output (default is ANSI).
	/unicode	Uses Unicode for output.
	/noil	Suppresses MSIL assembly code output.
	/header	Includes file header information.
	/item:class [::method sig]	Disassembles only the specific class or method (.exe, .dll, .obj, .lib files; console output only if /OUT is specified).
	/objectfile: filename	Shows metadata of a specific object file in a library (.lib files; console output only if /OUT is specified).

Category	Option	Description
Advanced options	/adv	Enables advanced command-line options and menu items. This option must precede other options in this group.
	/stats	Includes statistics on the image.
	/classlist	Includes a list of the classes defined in the module.
	/all	Combines /header, /bytes, /stats, /classlist, and /tokens options.
	/metadata: specifier	Shows metadata information; specifier can be MDHEADER (metadata header information and sizes), HEX (more things in hex as well as text), CSV (header sizes in CSV format), UNREX (unresolved externals), VALIDATE (validate the metadata for consistency).
Miscellaneous	/? or /help	Shows a summary of command-line switches.
	/adv /?	Shows a summary of standard and advanced command-line switches.
	/nobar	Suppresses disassembly progress bar popup.

The Native Image Generator Utility (NGEN)

The NGEN utility is what many articles and books written about beta versions of the .NET Framework refer to as the *pre-JIT compiler*. This little but useful utility works a bit like a traditional compiler in that it transforms the MSIL language in an assembly into native code that can be fed directly to the CPU. However, it differs in several important ways from a traditional compiler:

- The native code version of an assembly is stored in a special native code cache that you can't access directly. For example, you can't ship this native compiled version to your customers as you would do with the EXE files produced by Visual Basic 6.
- The .NET runtime uses the native compiled code only if the binding process would pick up the original MSIL assembly. In other words, if you use NGEN to compile version 1.0.0.0 of a given DLL, the native code version is used only when a .NET application requests that specific version of the DLL. In all other cases, the binding mechanism looks for the specific version requested by the application, and the native code version will be ignored. (But of course, you can use the application's configuration file to redirect all requests to version 1.0.0.0.) You must use NGEN for all the subsequent versions of the assembly to ensure that the native code version is always used, regardless of the requested version.

- The native code that NGEN generates is affected by the current application configuration, including binding policy, security policy, and development environment settings (such as debugging and profiling). If the configuration changes—for example, because you provide a custom configuration file that specifies a different binding policy—the runtime can't reuse the native compiled code and will load and JIT-compile the original assembly.
- Whenever you recompile the original source code and generate a new MSIL assembly with the same versions and similar attributes as one in the native-code cache, the new version becomes the active assembly and the one in the native-code cache is discarded. Hence, you need to rerun NGEN whenever you recompile an assembly to ensure that the native code version is used.
- Files in the native image cache depend on the runtime version. If you install a new version of the .NET runtime—for example, a Service Pack—you must rerun NGEN for all the assemblies you mean to precompile as native code.

As you might remember from Chapter 1, the JIT compiler transforms IL code into native code when your application invokes a method for the first time and then always uses the native code. The NGEN utility performs this transforming step before execution and can therefore reduce the start-up time of applications that perform a lot of method calls when they run. (Windows Forms applications can greatly benefit from this utility, for example.) However, the NGEN utility can't apply a few advanced optimization techniques that are possible with the JIT compiler—for example, cross-assembly optimization or code inlining based on how frequently a method is invoked—so you might even end up with code that's *slower* than standard JIT-compiled code.

Using NGEN is really easy because you just pass the list of assemblies to be compiled on its command line:

```
ngen MyApp.exe FirstLib.dll SecondLib.dll
```

You can also specify an assembly by its name (and optionally version, culture, and public key token), which makes sense for shared assemblies already installed in the GAC:

```
ngen
MyApp,Ver=1.0.0.0,Culture="us",PublicKeyToken=23a123bef89123ca
```

You can use the `/delete` option to delete a specific assembly and the `/show` option to display all the assemblies in the native image cache. (If you've never run NGEN on your assemblies, the latter command displays the names of a few precompiled assemblies in the .NET runtime.)

Three options let you create native code images that can be debugged or profiled. (By default, compiled assemblies generated by NGEN don't work with debuggers or profilers.) The `/debug` option generates an image that works with a debugger, `/debugopt` generates an image that works with a debugger in optimized debugged mode, and `/prof` generates an image that works with a profiler that requires instrumented code. (Check your profiler's documentation to see whether it requires this option.) In practice, you will rarely use these options because it's easier to debug and profile assemblies in plain MSIL code.