Deploying .NET Applications

Learning MSBuild and ClickOnce

Sayed Y. Hashimi and
Sayed Ibrahim Hashimi

Apress®

Deploying .NET Applications: Learning MSBuild and ClickOnce

ISBN-13 (pbk): 978-1-59059-652-4

ISBN-10 (pbk): 1-59059-652-8

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Jonathan Hassell
Technical Reviewer: Bart De Smet
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Jason Gilmore, Jonathan Gennick, Jonathan Hassell, James Huddleston, Chris Mills, Matthew Moodie, Dominic Shakeshaft, Jim Sumser, Keir Thomas, Matt Wade
Project Manager: Richard Dal Porto
Copy Edit Manager: Nicole LeClerc
Copy Editor: Kim Wimpsett
Assistant Production Director: Kari Brooks-Copony
Production Editor: Ellie Fountain
Compositor: Kinetic Publishing Services, LLC
Proofreader: Dan Shaw
Indexer: Brenda Miller
Artist: Kinetic Publishing Services, LLC
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Source Code section.

CHAPTER 2

The Unified Build Engine: MSBuild

Software systems have moved from stand-alone applications installed on single machines to large, distributed applications hosted over a network of machines. To create executables of stand-alone applications, you opened a command prompt and executed a few commands to convert the source code into executables. To deploy these applications, you took the generated executables and stored them on floppy disks and "sneakernetted" the application to the clients. The distributed systems of today are considerably more complex; they are much larger and so are their user bases. To build and deploy these systems, you must use predefined processes, along with automated build and deployment tools, to ensure reliability and repeatability. For example, most organizations define what are known as *pipelines*, and applications are built and deployed automatically to a pipeline for testing and verification. An application starts in an integration environment, then moves to a staging environment, and finally moves to production. At each point, tests are automatically executed in an attempt to ensure quality.

At the heart of a build and deployment process is a build and release tool. That is, one tool is responsible for getting, building, and then deploying these applications automatically. Moreover, it is the same tool that is used to migrate the applications from one environment (for example, staging) to another (for example, production).

Just as software development tools have evolved, so have the tools that are used to build and deploy them. Why do you need build tools? Why aren't scripts or batch files sufficient? Well, build tools are a necessary component of application development now because the steps for building software have increased in complexity. Previously, builds required simpler steps, such as copying and moving files, in order to perform the build. Now many applications are using third-party libraries and require more complex tasks, such as file signing and incremental building (more on this in Chapter 4). This increase in complexity has given rise to the need for build-specific tools. In other words, people started by building their software, somewhat manually, and realized they needed a repeatable process. Tools such as Make, NMake, Ant/NAnt, and Jam (among others) can create a repeatable process.

In this chapter, we will discuss MSBuild, but we will also highlight some of the most popular build tools and build systems used in the recent past. Note that a *build tool* is a component or application whose sole responsibility is to take source code and produce binaries (for example,

machine code, Java bytecode, Microsoft Intermediate Language [MSIL], and so on). A *build system* is a collection of build-related tools that together offer facilities to build, deploy, configure, and test solutions. Build tools and build systems are often packaged as part of popular software development systems.

Introducing Build Tools and Systems

In the following sections, we will describe some of the popular build tools and build systems.

Make-Style Build Tools

Several build tools are extensions of the original BSD Make. The most popular of the extensions include GNU Make, NMake, OPus Make, Jam, and Cook. With all of these tools, you place a file, usually called a *makefile*, near the source code that describes what needs to be built. To do a build, users usually enter `make` in a Unix shell or Windows Command window.

GNU Make

GNU Make is probably the most popular build tool on the Unix platform. GNU Make obtained its popularity because of the vast number of extensions it made to BSD Make. GNU Make is a part of the GNU toolset and, thus, is still alive and supported. This build tool is distributed under the GNU open source license.

NMake

NMake was originally developed by AT&T Laboratories as an open source project. Recently it has been extended by Lucent Technologies and is packaged as part of a commercial product called Lucent nmake Product Builder. Microsoft also has a version it calls Microsoft Program Maintenance Utility (`NMAKE.EXE`). The versions produced by AT&T and Lucent are compatible with the original BSD Make; the version produced by Microsoft is not.

OPus Make

OPus Make was a popular build tool in the 90s. This tool became popular for its multiplatform support and rather lengthy list of features, including support for logical operators in conditional expressions, regular expression substitutions, a rich set of directives, and more.

Jam and Cook

GNU Make, NMake, and OPus Make were all extensions of Make but use the same style used by the original BSD Make. For example, they all relied on the makefile and added features (such as macros). Jam and Cook are variants of Make that don't rely on the makefile but use another variation of a text file to define what needs to be built. The interesting feature of these two products is that they support parallel builds while avoiding recursion.

GBS

The GNU Build System (GBS) is a suite of build tools that, together, provide facilities to build, configure, and release software systems. GBS is popular in the open source community because it provides a feature that places an abstraction layer over the build platform. GBS is mostly used within the open source community.

CMake and QMake

Cross Platform Make (CMake) and Qtopia Build System (QMake) are build systems, not build tools, that decided to piggyback on the popularity of Make. These build systems have a lot of similarities in that they both market their portability features and support various build tools. CMake offers a GUI and a command-line interface; QMake supports only a command-line interface.

Ant/NAnt

The build tools described in the previous sections all use a build description file (for example, a makefile). Most rely on either cryptic commands or full-blown programming languages to describe what needs to be built. The problem with describing builds using commands and/or languages is obviously that the builds are difficult to create and maintain.

Ant is a cross-platform build tool popular in the Java community. Ant discarded the idea of using commands or programming languages to describe builds and instead uses XML-based configuration files. Build steps are described using something called *tasks*, which are implemented with the Java programming language (rather than by writing scripts) and are grouped under a *target*. This is an example Ant file:

```
<project>
    <target name="compile">
        <mkdir dir="build/classes"/>
        <javac srcdir="src" destdir="build/classes"/>
    </target>

    <target name="jar">
        <mkdir dir="build/jar"/>
        <jar destfile="build/jar/HelloWorld.jar" basedir="build/classes">
            <manifest>
                <attribute name="Main-Class" value="sayed.HelloWorld"/>
            </manifest>
        </jar>
    </target>

    <target name="run">
        <java jar="build/jar/HelloWorld.jar" fork="true"/>
    </target>
</project>
```

To perform a build using Ant, you place a file named `build.xml` next to your source tree and enter `ant` at the command prompt. Ant is command-line driven; however, several popular extensions offer GUIs for Ant.

As far as NAnt goes, NAnt is a port of Ant to the .NET platform. NAnt maintains the same concepts of Ant. That is, NAnt also uses an XML-based build file and defines features such as targets, tasks, and so on.

We'll now begin discussing the next-generation build tool: MSBuild.

Introducing MSBuild

MSBuild is Microsoft's solution to the automated build problem. MSBuild allows you to perform all the necessary steps to properly build your .NET applications. MSBuild provides this functionality transparently.

In prior versions of Visual Studio, the build process was, for the most part, hidden to the user. You could, for example, supplement your build with pre- and post-events, but you could not change *how* the build occurred. In Visual Studio 2005, you have this feature! Visual Studio 2005 uses MSBuild to build your solutions. Microsoft has exposed and defined this build process as part of the MSBuild schema.

Visual Studio uses your project file as input to MSBuild. In this example, we will show how to create a simple project file using Visual Studio. You can use this file to understand some of the key elements of MSBuild. It will also serve as a point of extension to explore some of the other features of MSBuild. Although MSBuild supports many application types, we will show how to create a simple Windows Forms application.

To create this simple project, follow these steps:

1. Start Visual Studio, and create a new C# Windows application named **MSBuild1**.
2. Accept the defaults, and click OK.
3. In the Form Designer, drag and drop a new label onto the form.
4. Set the text of the label to **MSBuild demo**.

The form should look like Figure 2-1.

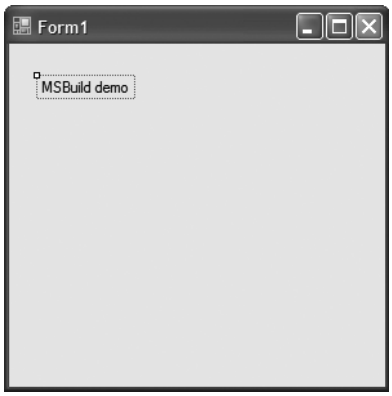

Figure 2-1. *Sample Windows application*

Open the project file (`MSBuild1.csproj`) in your favorite XML or text editor. You'll notice that the root element is the `Project` element. Beneath this you'll see three element types in this file:

- `PropertyGroup`
- `ItemGroup`
- `Import`

A few other elements could be present at this level, but we will discuss those elements as you get to them. You'll see an element that is commented out as well: the `Target` element. We will discuss this important piece of the MSBuild file later in this chapter. The `PropertyGroup` element is a container for defined properties. Similarly, the `ItemGroup` element is a container for defined items. The `Import` tag allows you to import other MSBuild files into the current project. We will examine how this will affect your files later in this chapter.

An MSBuild file has four main elements: properties, items, targets, and tasks. A *property* defines a value associated with a name. Simply put, it is a key/value pair. In this project file, you'll find many properties defined:

```
<DebugSymbols>true</DebugSymbols>
<DebugType>full</DebugType>
<Optimize>false</Optimize>
<OutputPath>bin\Debug\</OutputPath>
```

You'll find these defined under the `<PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">` tag. You can change these values directly from Visual Studio by using the Configuration drop-down list.

Items are another crucial part of the MSBuild project file. When performing a build, many steps must reference a file or a set of files. In MSBuild this is usually accomplished through items. An *item* is a named reference to a file or to many files. These items contain associated metadata, such as the full path or filename. Throughout the discussion of MSBuild, we will discuss items in more depth.

A *target* is a container for related tasks that will be executed sequentially. Besides containing tasks, a target can be given a set of dependent targets and a list of inputs and outputs. *Incremental building*, which is discussed in more detail in Chapter 4, is the process of skipping unnecessary steps. This is driven completely by the associated input and output files for a target. When you invoke MSBuild, you must specify a target that is to be executed; after that, MSBuild will perform a dependency analysis to determine exactly what other targets need to be executed as well.

A *task* is a unit of work in MSBuild. For example, `Copy` or `LocateRequiredAssemblies` could be defined as a task. In this sample, you will not find any tasks defined. This is because this sample utilizes predefined tasks to complete the build. (We will cover the predefined task later in this chapter in the "Predefined Tasks" section.) Tasks must be located within a `Target` element. A `Target` element consists of a group of related tasks that are executed sequentially. Possible targets are `PrepareForDeployment` and `CopyToServers`.

As mentioned, this project uses predefined tasks to accomplish the build. You may be wondering where these tasks are actually defined. To answer that question, scroll toward the bottom of the project file, and you'll find the following declaration: `<Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />`. This import statement is using a property, `MSBuildBinPath`, to specify where to find the `Microsoft.CSharp.targets` file, which contains many predefined tasks. The `Microsoft.CSharp.targets` file is located in the `%windir%\Microsoft.NET\Framework\v2.0.50727\` directory. By using the `$()` syntax, the property `MSBuildBinPath` is evaluated, and its value replaces the reference. This property is a reserved property whose value is the location of `msbuild.exe`. A few other reserved properties exist, which we will discuss in the next section.

Properties

As stated, a property is a simple key/value pair. Let's examine another property definition from the `MSBuild.csproj` file: `<RootNamespace>MSBuild1</RootNamespace>`. This property is defined in the first `PropertyGroup` element. If you needed to reference this property somewhere else in the MSBuild file, you would simply use the `$()` notation. For instance, you would use `$(RootNamespace)`.

Refer to the property declaration again: `<PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">`. Notice the `Condition` attribute; every MSBuild element has an optional `Condition` attribute. If this condition evaluates to `true`, then the element is processed; otherwise, it is ignored. Table 2-1 summarizes the basic condition syntax.

Table 2-1. *Property Conditions*

Symbol	Description
`==`	Checks for equality; returns `true` if both have the same value.
`!=`	Checks for inequality; returns `true` if both don't have the same value.
`Exists`	Checks for existence of a file. You provide the file path as an argument, such as in `Exists(MSBuildDemo.txt)`. This will return `true` if `MSBuildDemo.txt` is present.
`!Exists`	Checks for the nonexistence of a file. You use this condition in a similar manner as the `Exists` condition.

The `<Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>` property will not be executed unless the `Configuration` property has not already been set. In the next section, you'll see how you can use these properties in tasks that you create. You may find a number of reserved properties helpful in your MSBuild project. Table 2-2 lists their names, their descriptions, and the values that are returned for the project you will start shortly. In the "Targets" section, you will create a target to print the values for these properties.

Table 2-2. *Reserved Properties*

Name	Description	Example
`MSBuildBinPath`	Full path of the .NET Framework MSBuild `bin` directory.	`%windir%\Microsoft.NET\Framework\v2.0.50727`
`MSBuildExtensionsPath`	Full path to the `MSBuild` folder located in the `Program Files` directory. This is a nice location to keep other MSBuild files that the current file references.	`C:\Program Files\MSBuild`
`MSBuildProjectDefaultTargets`	The value for the `DefaultTargets` attribute that is in the `Project` element.	`Build`
`MSBuildProjectDirectory`	Full path to the location of project file.	`C:\MSBuild\MSBuild1\MSBuild1`
`MSBuildProjectExtension`	Extension of the project filename, including the initial dot.	`.csproj`

Name	Description	Example
MSBuildProjectFile	Filename of the project file, including extension.	MSBuild1.csproj
MSBuildProjectFullPath	Full path to the project file, including the filename.	C:\MSBuild\MSBuild1\MSBuild1\MSBuild1.csproj
MSBuildProjectName	Name of the MSBuild project file, excluding the file extension.	MSBuild1

Targets

As mentioned, *targets* are containers for related tasks that will be executed sequentially. Some example targets are `Build`, `Deploy`, and `SetupEnvironment`. Let's examine the parts of a target:

```
<Target
    Name="SampleTarget"
    Inputs="SampleInput"
    Outputs="SampleOutput"
    DependsOnTargets="DependentTarget"
>
    <Message Text="SampleTarget executed, SampleInput: @(SampleInput)" />
</Target>
<Target Name="DependentTarget">
    <Message Text="DependentTarget executed" />
</Target>
```

Each target has a `Name` attribute that is required to be a nonempty string. This name is how you will refer to the target. Additionally, a target can have inputs; if you declare a target to have inputs, then it must have outputs as well. The purpose of the inputs/outputs is to facilitate incremental builds. That is, if a portion of your build does not need to be reexecuted, then it will not be. In a build process, if you had a `Target` defined as `CopyResources`, it may take as an input a list of files containing the location on disk of all external resources. The corresponding output may be the desired location of these resources. When MSBuild encounters this target, it will compare the time stamps of these files to each other. If it is not necessary to reexecute that `CopyResources` target, then it will be skipped.

The `DependsOnTarget` parameter is a list of targets, separated by semicolons, that are required to be run before this target executes. It is important to note at this time that during a build a target will be executed only once. So, if you had two targets that both depended on a common target, that one target will not be executed twice but only once. Now you will examine how you can get started executing some targets that employ some of the predefined tasks.

Executing and Creating Targets

In this section, we will discuss the process involved in executing and creating targets. You can utilize many predefined targets in your builds. (Many predefined tasks also exist; we will wait to cover those in the "Predefined Tasks" section.) To emphasize the detachment of MSBuild from Visual Studio, we will be executing the targets strictly from the command line. The first step you'll want to perform is to open the previously created project file in your favorite text editor. Then, open the Visual Studio 2005 command prompt from the Start menu. From the

command prompt, navigate to the directory in which your `MSBuild1.csproj` file is located. (Note that MSBuild is also capable of processing solution files, despite that solutions files are not in MSBuild format.) From there, execute the following command: `>msbuild`. This will execute the default target on the only project file residing in that directory. Your output should look something like Figure 2-2.

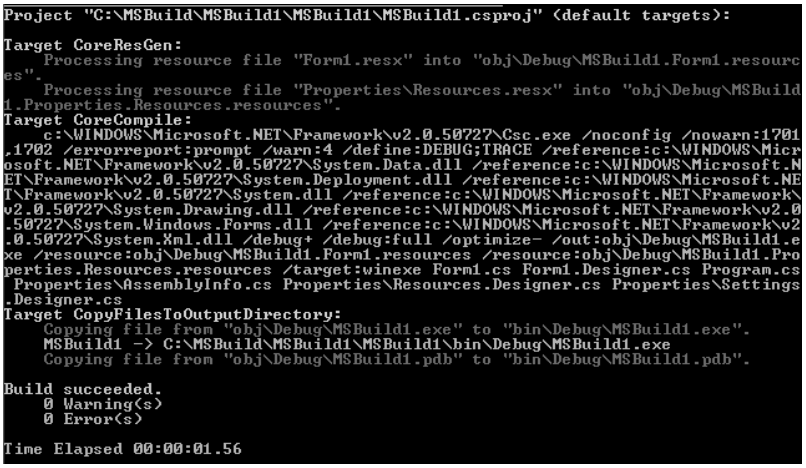

Figure 2-2. *Output from* `msbuild.exe` *on the default target*

You specify the default target in the root `Project` element of the project file. In this case, you have `<Project DefaultTargets="Build" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">`. If you invoke MSBuild on this project file without specifying the target, then the `Build` project will be executed. This target is a predefined target and is not included in your project file. Actually, no targets are defined in this file. This file has an import statement that imports `Microsoft.CSharp.targets`, and that file imports `Microsoft.Common.targets`. These two files define many targets, and these are the predefined targets. We will discuss this in more detail in the "Predefined Targets" section.

Now let's inject the two targets discussed earlier into this project file. You can place them anywhere in the file as long as they are defined as child elements of the `Project` tag. As a reminder, the two targets are as follows:

```
<Target
    Name="SampleTarget"
    Inputs="SampleInput"
    Outputs="SampleOutput"
    DependsOnTargets="DependentTarget"
>
    <Message Text="SampleTarget executed, SampleInput: @(SampleInput) " />
</Target>
<Target Name="DependentTarget">

    <Message Text="DependentTarget executed" />
</Target>
```

At this point, save this file as `MSBuild1_rev2.csproj` so you have a backup of the project file in the same directory. The previous example executed the only project in the directory. If two or more project files exist, then you must specify which one to execute. To do this, you can supply the name of the project to build as a command-line parameter. To build this project, execute the following command: `>msbuild MSBuild1_rev2.csproj`. Following this, you may see similar output as you did when you built the previous project, or you will see that many targets were skipped. This depends on whether your source files have changed since your last build. Skipping up-to-date targets is called *incremental building* and is a core aspect of MSBuild; we will discuss this in the "Predefined Targets" section and in more depth in Chapter 4. If you'd like to see it build again, you can invoke `>msbuild MSBuild1_rev2.csproj /t:Clean;build`. This will clean out the previously built files and then build the project. Notice that the target names are case-insensitive. To execute the target, you simply execute `>msbuild MSBuild1_rev2.csproj /t:SampleTarget`. Figure 2-3 shows the output.

```
Project "C:\MSBuild\MSBuild1\MSBuild1\MSBuild1_rev2.csproj" (SampleTarget target
(s)):

Target DependentTarget:
    DependentTarget executed
Target SampleTarget:
    SampleTarget executed, SampleInput:

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:00.03
```

Figure 2-3. *Output from the execution of* `SampleTarget`

Now that you have started specifying some parameters for `msbuild.exe`, you may be interested in what other options are available. Table 2-3 summarizes those options.

Table 2-3. `msbuild.exe` *Command-Line Parameters*

Switch	Short	Description
`/help`	`/?`	Displays usage for `msbuild.exe`.
`/nologo`		Inhibits the copyright message when `msbuild.exe` is executed.
`/version`	`/ver`	Displays the version of `msbuild.exe`.
`@file`		Allows you to pass command-line parameters to `msbuild.exe` from the file specified.
`/noautoresponse`	`/noautorsp`	Allows you to specify to not automatically include the `msbuild.rsp` file. This file can specify command-line arguments for MSBuild. If it is present, it will be consumed, unless you set this flag. If you have long command-line arguments, this is the suggested manner to pass them to MSBuild.

Continued

Table 2-3. *Continued*

Switch	Short	Description
/target:<target>	/t	Specifies which targets should be executed. Targets are declared in a semicolon-separated list.
/property:<n>=<v>	/p	Allows you to set properties for the build. If a property is specified that exists in the project file, then this value will take precedence.
/logger:<logger>		Specifies the logger used to capture and records MSBuild events as they occur.
/verbosity:<level>	/v	Sets the type of information MSBuild will output. Possible values include d (detailed), diag (diagnostic), m (minimal), q (quiet), and n (normal).
/consoleloggerparameters <parameters>	/clp	Passes parameters to the console logger for MSBuild.
/noconsolelogger	/noconlog	Turns off logging to the console.
/validate	/val	Validates the MSBuild project file with the MSBuild schema file in use.
/validate:<schema>	/val	Validates the MSBuild project file with the MSBuild schema file specified.

From the output of the previous example in Figure 2-3, did you notice that the dependent target executed first? As mentioned, a target will execute only once during the build process. To demonstrate this, add the following target to your MSBuild project file:

```
<Target
    Name="DependsAgain"
    DependsOnTargets="DependentTarget"
>
    <Message Text="DependsAgain has executed"/>
</Target>
```

Now invoke MSBuild with the following command: `C:\MSBuild\MSBuild1\MSBuild1> msbuild MSBuild1_rev2.csproj /t:SampleTarget;DependsAgain`. Figure 2-4 shows the output.

```
Project "C:\MSBuild\MSBuild1\MSBuild1\MSBuild1_rev2.csproj" (SampleTarget;Depend
sAgain target(s)):

Target DependentTarget:
    DependentTarget executed
Target SampleTarget:
    SampleTarget executed, SampleInput:
Target DependsAgain:
    DependsAgain has executed

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:00.01
```

Figure 2-4. `msbuild.exe` *output for the target* `SampleTarget;DependsAgain`

As you can see, DependentTarget seems to have executed only once, before the execution of SampleTarget. The output presented in Figure 2-4 does not make it obvious that this target was skipped the second time. If you use the command-line parameter /v:d or /v:diag, it will explicitly state that this target was indeed skipped. Previously it was mentioned that a Target will be described to print the values for the reserved properties; the specification for that target is as follows:

```
<Target Name="PrintReservedProperties">
   <Message Text="MSBuildProjectDirectory      : ➥
$(MSBuildProjectDirectory)" />
   <Message Text="MSBuildProjectFile           : ➥
$(MSBuildProjectFile)" />
   <Message Text="MSBuildProjectExtension      : ➥
$(MSBuildProjectExtension)" />
   <Message Text="MSBuildProjectFullPath       : ➥
$(MSBuildProjectFullPath)" />
   <Message Text="MSBuildProjectName           : ➥
$(MSBuildProjectName)" />
   <Message Text="MSBuildBinPath               : ➥
$(MSBuildBinPath)" />
   <Message Text="MSBuildProjectDefaultTargets : ➥
$(MSBuildProjectDefaultTargets)" />
   <Message Text="MSBuildExtensionsPath        : ➥
$(MSBuildExtensionsPath)" />
</Target>
```

After you add this to the project file, you can invoke it with the following command: >msbuild MSBuild1.csproj /t:PrintReservedProperties. Note this was added to the original version of the project file, not the MSBuild1_rev2.csproj file.

In Figure 2-5, you can see the values for the reserved properties that are available with MSBuild. Now we will explain some predefined targets.

```
Project "C:\MSBuild\MSBuild1\MSBuild1\MSBuild1_rev2.csproj" (PrintReservedProper
ties target(s)):

Target PrintReservedProperties:
    MSBuildProjectDirectory      : C:\MSBuild\MSBuild1\MSBuild1
    MSBuildProjectFile           : MSBuild1_rev2.csproj
    MSBuildProjectExtension      : .csproj
    MSBuildProjectFullPath       : C:\MSBuild\MSBuild1\MSBuild1\MSBuild1_rev2.cs
proj
    MSBuildProjectName           : MSBuild1_rev2
    MSBuildBinPath               : C:\WINDOWS\Microsoft.NET\Framework\v2.0.50215

    MSBuildProjectDefaultTargets : Build
    MSBuildExtensionsPath        : C:\Program Files\MSBuild

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:00.02
```

Figure 2-5. *Output for reserved properties*