Contents

CreateCSharpManifestResourceName Task

CreateItem Task

CreateProperty Task

CreateVisualBasicManifestResourceName Task

Csc Task

Delete Task

DownloadFile Task

Error Task

Exec Task

FindAppConfigFile Task

FindInList Task

FindUnderPath Task

FormatUrl Task

FormatVersion Task

GenerateApplicationManifest Task

GenerateBootstrapper Task

GenerateDeploymentManifest Task

GenerateResource Task

GenerateTrustInfo Task

GetAssemblyIdentity Task

GetFileHash Task

GetFrameworkPath Task

GetFrameworkSdkPath Task

GetReferenceAssemblyPaths Task

LC Task

MakeDir Task

Message Task

Move Task

MSBuild Task

ReadLinesFromFile Task

RegisterAssembly Task

RemoveDir Task

MSBuild

10/21/2019 • 9 minutes to read • Edit Online

The Microsoft Build Engine is a platform for building applications. This engine, which is also known as MSBuild, provides an XML schema for a project file that controls how the build platform processes and builds software. Visual Studio uses MSBuild, but it doesn't depend on Visual Studio. By invoking *msbuild.exe* on your project or solution file, you can orchestrate and build products in environments where Visual Studio isn't installed.

Visual Studio uses MSBuild to load and build managed projects. The project files in Visual Studio (*.csproj*, *.vbproj*, *.vcxproj*, and others) contain MSBuild XML code that executes when you build a project by using the IDE. Visual Studio projects import all the necessary settings and build processes to do typical development work, but you can extend or modify them from within Visual Studio or by using an XML editor.

For information about MSBuild for C++, see MSBuild (C++).

The following examples illustrate when you might run builds by using an MSBuild command line instead of the Visual Studio IDE.

- Visual Studio isn't installed. (download MSBuild without Visual Studio)
- You want to use the 64-bit version of MSBuild. This version of MSBuild is usually unnecessary, but it allows MSBuild to access more memory.
- You want to run a build in multiple processes. However, you can use the IDE to achieve the same result on projects in C++ and C#.
- You want to modify the build system. For example, you might want to enable the following actions:
  - Preprocess files before they reach the compiler.
  - Copy the build outputs to a different place.
  - Create compressed files from build outputs.
  - Do a post-processing step. For example, you might want to stamp an assembly with a different version.

You can write code in the Visual Studio IDE but run builds by using MSBuild. As another alternative, you can build code in the IDE on a development computer but use an MSBuild command line to build code that's integrated from multiple developers.

> **NOTE**
> You can use Team Foundation Build to automatically compile, test, and deploy your application. Your build system can automatically run builds when developers check in code (for example, as part of a Continuous Integration strategy) or according to a schedule (for example, a nightly Build Verification Test build). Team Foundation Build compiles your code by using MSBuild. For more information, see Azure Pipelines.

This topic provides an overview of MSBuild. For an introductory tutorial, see Walkthrough: Using MSBuild.

Use MSBuild at a command prompt

To run MSBuild at a command prompt, pass a project file to *MSBuild.exe*, together with the appropriate

command-line options. Command-line options let you set properties, execute specific targets, and set other options that control the build process. For example, you would use the following command-line syntax to build the file *MyProj.proj* with the `Configuration` property set to `Debug`.

```
MSBuild.exe MyProj.proj -property:Configuration=Debug
```

For more information about MSBuild command-line options, see Command-line reference.

> **IMPORTANT**
> Before you download a project, determine the trustworthiness of the code.

Project file

MSBuild uses an XML-based project file format that's straightforward and extensible. The MSBuild project file format lets developers describe the items that are to be built, and also how they are to be built for different operating systems and configurations. In addition, the project file format lets developers author reusable build rules that can be factored into separate files so that builds can be performed consistently across different projects in the product.

The following sections describe some of the basic elements of the MSBuild project file format. For a tutorial about how to create a basic project file, see Walkthrough: Creating an MSBuild project file from scratch.

Properties

Properties represent key/value pairs that can be used to configure builds. Properties are declared by creating an element that has the name of the property as a child of a PropertyGroup element. For example, the following code creates a property named `BuildDir` that has a value of `Build`.

```
<PropertyGroup>
    <BuildDir>Build</BuildDir>
</PropertyGroup>
```

You can define a property conditionally by placing a `Condition` attribute in the element. The contents of conditional elements are ignored unless the condition evaluates to `true`. In the following example, the `Configuration` element is defined if it hasn't yet been defined.

```
<Configuration  Condition=" '$(Configuration)' == '' ">Debug</Configuration>
```

Properties can be referenced throughout the project file by using the syntax $(<PropertyName>). For example, you can reference the properties in the previous examples by using `$(BuildDir)` and `$(Configuration)`.

For more information about properties, see MSBuild properties.

Items

Items are inputs into the build system and typically represent files. Items are grouped into item types, based on user-defined item names. These item types can be used as parameters for tasks, which use the individual items to perform the steps of the build process.

Items are declared in the project file by creating an element that has the name of the item type as a child of an ItemGroup element. For example, the following code creates an item type named `Compile`, which includes two files.

```
<ItemGroup>
    <Compile Include = "file1.cs"/>
    <Compile Include = "file2.cs"/>
</ItemGroup>
```

Item types can be referenced throughout the project file by using the syntax @(<ItemType>). For example, the item type in the example would be referenced by using `@(Compile)`.

In MSBuild, element and attribute names are case-sensitive. However, property, item, and metadata names are not. The following example creates the item type `Compile`, `comPile`, or any other case variation, and gives the item type the value "one.cs;two.cs".

```
<ItemGroup>
  <Compile Include="one.cs" />
  <comPile Include="two.cs" />
</ItemGroup>
```

Items can be declared by using wildcard characters and may contain additional metadata for more advanced build scenarios. For more information about items, see Items.

Tasks

Tasks are units of executable code that MSBuild projects use to perform build operations. For example, a task might compile input files or run an external tool. Tasks can be reused, and they can be shared by different developers in different projects.

The execution logic of a task is written in managed code and mapped to MSBuild by using the UsingTask element. You can write your own task by authoring a managed type that implements the ITask interface. For more information about how to write tasks, see Task writing.

MSBuild includes common tasks that you can modify to suit your requirements. Examples are Copy, which copies files, MakeDir, which creates directories, and Csc, which compiles Visual C# source code files. For a list of available tasks together with usage information, see Task reference.

A task is executed in an MSBuild project file by creating an element that has the name of the task as a child of a Target element. Tasks typically accept parameters, which are passed as attributes of the element. Both MSBuild properties and items can be used as parameters. For example, the following code calls the MakeDir task and passes it the value of the `BuildDir` property that was declared in the earlier example.

```
<Target Name="MakeBuildDirectory">
    <MakeDir  Directories="$(BuildDir)" />
</Target>
```

For more information about tasks, see Tasks.

Targets

Targets group tasks together in a particular order and expose sections of the project file as entry points into the build process. Targets are often grouped into logical sections to increase readability and to allow for expansion. Breaking the build steps into targets lets you call one piece of the build process from other targets without copying that section of code into every target. For example, if several entry points into the build process require references to be built, you can create a target that builds references and then run that target from every entry point where it's required.

Targets are declared in the project file by using the Target element. For example, the following code creates a target named `Compile`, which then calls the Csc task that has the item list that was declared in the earlier example.

```
<Target Name="Compile">
    <Csc Sources="@(Compile)" />
</Target>
```

In more advanced scenarios, targets can be used to describe relationships among one another and perform dependency analysis so that whole sections of the build process can be skipped if that target is up-to-date. For more information about targets, see Targets.

Build logs

You can log build errors, warnings, and messages to the console or another output device. For more information, see Obtaining build logs and Logging in MSBuild.

Use MSBuild in Visual Studio

Visual Studio uses the MSBuild project file format to store build information about managed projects. Project settings that are added or changed by using the Visual Studio interface are reflected in the *.*proj* file that's generated for every project. Visual Studio uses a hosted instance of MSBuild to build managed projects. This means that a managed project can be built in Visual Studio or at a command prompt (even if Visual Studio isn't installed), and the results will be identical.

For a tutorial about how to use MSBuild in Visual Studio, see Walkthrough: Using MSBuild.

Multitargeting

By using Visual Studio, you can compile an application to run on any one of several versions of the .NET Framework. For example, you can compile an application to run on the .NET Framework 2.0 on a 32-bit platform, and you can compile the same application to run on the .NET Framework 4.5 on a 64-bit platform. The ability to compile to more than one framework is named multitargeting.

These are some of the benefits of multitargeting:

- You can develop applications that target earlier versions of the .NET Framework, for example, versions 2.0, 3.0, and 3.5.
- You can target frameworks other than the .NET Framework, for example, Silverlight.
- You can target a *framework profile*, which is a predefined subset of a target framework.
- If a service pack for the current version of the .NET Framework is released, you could target it.
- Multitargeting guarantees that an application uses only the functionality that's available in the target framework and platform.

For more information, see Multitargeting.

See also

TITLE	DESCRIPTION
Walkthrough: Creating an MSBuild project file from scratch	Shows how to create a basic project file incrementally, by using only a text editor.
Walkthrough: Using MSBuild	Introduces the building blocks of MSBuild and shows how to write, manipulate, and debug MSBuild projects without closing the Visual Studio IDE.

TITLE	DESCRIPTION
MSBuild concepts	Presents the four building blocks of MSBuild: properties, items, targets, and tasks.
Items	Describes the general concepts behind the MSBuild file format and how the pieces fit together.
MSBuild properties	Introduces properties and property collections. Properties are key/value pairs that can be used to configure builds.
Targets	Explains how to group tasks together in a particular order and enable sections of the build process to be called on the command line.
Tasks	Shows how to create a unit of executable code that can be used by MSBuild to perform atomic build operations.
Conditions	Discusses how to use the `Condition` attribute in an MSBuild element.
Advanced concepts	Presents batching, performing transforms, multitargeting, and other advanced techniques.
Logging in MSBuild	Describes how to log build events, messages, and errors.
Additional resources	Lists community and support resources for more information about MSBuild.

Reference

- MSBuild reference Links to topics that contain reference information.
- Glossary Defines common MSBuild terms.

What's new in MSBuild 16.0

3/14/2019 • 2 minutes to read • Edit Online

This article describes updated features and properties in MSBuild 16.0. For the detailed release notes (draft only), see MSBuild 16.0.

Changed path

MSBuild is installed in the *\Current* folder under each version of Visual Studio. For example, *C:\Program Files (x86)\Microsoft Visual Studio\Current\Enterprise\MSBuild*. You can also use the following PowerShell module to locate MSBuild: vssetup.powershell.

Changed properties

The following MSBuild properties have been updated due to the new version number.

- `MSBuildToolsVersion` for this version of the tools is "Current". The assembly version is the same as in Visual Studio 2017, which is 15.1.0.0.
- `VisualStudioVersion` for this version of the tools is "16.0"

Updates

MSBuild (and Visual Studio) now targets .NET Framework 4.7.2. If you wish to use new MSBuild API features, your assembly must also upgrade, but existing code will continue to work.

See also

- MSBuild

What's new in MSBuild 15

9/11/2019 • 2 minutes to read • Edit Online

MSBuild is now available as part of the .NET Core SDK and can build .NET Core projects on Windows, macOS, and Linux.

Changed path

MSBuild is now installed in a folder under each version of Visual Studio. For example, *C:\Program Files (x86)\Microsoft Visual Studio\2017\Enterprise\MSBuild*. You can also use the following PowerShell module to locate MSBuild: vssetup.powershell.

MSBuild is no longer installed in the Global Assembly Cache. To reference MSBuild programmatically, use NuGet packages. For more information, see Updating an existing application for MSBuild 15.0.

Changed properties

The following MSBuild properties have been updated due to the new version number.

- `MSBuildToolsVersion` for this version of the tools is 15.0. The assembly version is 15.1.0.0.
- `MSBuildToolsPath` no longer has a fixed location. By default, it is located in the *MSBuild\15.0\Bin* folder relative to the Visual Studio installation location, but the Visual Studio installation location can be changed at install time.
- `ToolsVersion` values are no longer set in the registry.
- The `SDK35ToolsPath` and `SDK40ToolsPath` properties point to the .NET Framework SDK that's packaged with this version of Visual Studio (for example, 10.0A for the 4.X tools).

Updates

- Project element has a new `SDK` attribute. Also the `Xmlns` attribute is now optional. For more information on the `SDK` attribute, see How to: Use MSBuild project SDKs, Packages, metapackages, and frameworks and Additions to the csproj format for .NET Core.
- Item element outside targets has a new `Update` attribute. Also, the restriction on the `Remove` attribute has been eliminated.
- *Directory.Build.props* is a user-defined file that provides customizations to projects under a directory. This file is automatically imported from *Microsoft.Common.props* unless the property `ImportDirectoryBuildTargets` is set to **false**. *Directory.Build.targets* is imported by *Microsoft.Common.targets*.
- Any metadata with a name that doesn't conflict with the current list of attributes can optionally be expressed as an attribute. For more information, see Item element.

New property functions

- `EnsureTrailingSlash` adds a trailing slash to a path if one doesn't already exist.
- `NormalizePath` combines path elements and ensures that the output string has the correct directory separator characters for the current operating system.
- `NormalizeDirectory` combines path elements, ensures a trailing slash, and ensures that the output string has the correct directory separator characters for the current operating system.

- `GetPathOfFileAbove` returns the path of the file immediately preceding this one. It is functionally equivalent to calling

```
<Import Project="$([MSBuild]::GetDirectoryNameOfFileAbove($(MSBuildThisFileDirectory), dir.props))\dir.props"
/>
```

See also

- MSBuild

MSBuild concepts

2/21/2019 • 2 minutes to read • Edit Online

MSBuild provides a basic XML schema that you can use to control how the build platform builds software. To specify the components in the build and how they are to be built, use these four parts of MSBuild: properties, items, tasks, and targets.

Related topics

TITLE	DESCRIPTION
MSBuild properties	Introduces properties and property collections. Properties are key/value pairs that you can use to configure builds.
MSBuild items	Introduces items and item collections. Items are inputs into the build system and typically represent files.
MSBuild targets	Explains how to group tasks together in a particular order and enable sections of the build process to be called on the command line.
MSBuild tasks	Shows how to create a unit of executable code that can be used by MSBuild to perform atomic build operations.
Comparing properties and items	Compares MSBuild properties and items. Both are used to pass information to tasks, evaluate conditions, and store values that can be referenced throughout the project file.
MSBuild special characters	Explains how to escape some characters that MSBuild reserves for special use in specific contexts.
Walkthrough: Creating an MSBuild project file from scratch	Shows how to create a basic project file incrementally, by using only a text editor.
Walkthrough: Using MSBuild	Introduces the building blocks of MSBuild and shows how to write, manipulate, and debug MSBuild projects without closing the Visual Studio integrated development environment (IDE).
MSBuild reference	Links to documents that contain reference information.
MSBuild	Presents an overview of the XML schema for a project file and shows how it controls processes that builds software.

MSBuild properties

4/23/2019 • 4 minutes to read • Edit Online

Properties are name-value pairs that can be used to configure builds. Properties are useful for passing values to tasks, evaluating conditions, and storing values that will be referenced throughout the project file.

Define and reference properties in a project file

Properties are declared by creating an element that has the name of the property as a child of a PropertyGroup element. For example, the following XML creates a property named `BuildDir` that has a value of `Build`.

```
<PropertyGroup>
    <BuildDir>Build</BuildDir>
</PropertyGroup>
```

Throughout the project file, properties are referenced by using the syntax $(<PropertyName>). For example, the property in the previous example is referenced by using $(BuildDir).

Property values can be changed by redefining the property. The `BuildDir` property can be given a new value by using this XML:

```
<PropertyGroup>
    <BuildDir>Alternate</BuildDir>
</PropertyGroup>
```

Properties are evaluated in the order in which they appear in the project file. The new value for `BuildDir` must be declared after the old value is assigned.

Reserved properties

MSBuild reserves some property names to store information about the project file and the MSBuild binaries. These properties are referenced by using the $ notation, just like any other property. For example, $(MSBuildProjectFile) returns the complete file name of the project file, including the file name extension.

For more information, see How to: Reference the name or location of the project file and MSBuild reserved and well-known properties.

Environment properties

You can reference environment variables in project files just as you reference reserved properties. For example, to use the `PATH` environment variable in your project file, use $(Path). If the project contains a property definition that has the same name as an environment property, the property in the project overrides the value of the environment variable.

Each MSBuild project has an isolated environment block: it only sees reads and writes to its own block. MSBuild only reads environment variables when it initializes the property collection, before the project file is evaluated or built. After that, environment properties are static, that is, each spawned tool starts with the same names and values.

To get the current value of environment variables from within a spawned tool, use the Property functions System.Environment.GetEnvironmentVariable. The preferred method, however, is to use the task parameter

EnvironmentVariables. Environment properties set in this string array can be passed to the spawned tool without affecting the system environment variables.

> **TIP**
> Not all environment variables are read in to become initial properties. Any environment variable whose name is not a valid MSBuild property name, such as "386", is ignored.

For more information, see How to: Use environment variables in a build.

Registry properties

You can read system registry values by using the following syntax, where `Hive` is the registry hive (for example, **HKEY_LOCAL_MACHINE**), `MyKey` is the key name, `MySubKey` is the subkey name, and `Value` is the value of the subkey.

```
$(registry:Hive\MyKey\MySubKey@Value)
```

To get the default subkey value, omit the `Value`.

```
$(registry:Hive\MyKey\MySubKey)
```

This registry value can be used to initialize a build property. For example, to create a build property that represents the Visual Studio web browser home page, use this code:

```
<PropertyGroup>
  <VisualStudioWebBrowserHomePage>
    $(registry:HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\14.0\WebBrowser@HomePage)
  </VisualStudioWebBrowserHomePage>
<PropertyGroup>
```

Global properties

MSBuild lets you set properties on the command line by using the **-property** (or **-p**) switch. These global property values override property values that are set in the project file. This includes environment properties, but does not include reserved properties, which cannot be changed.

The following example sets the global `Configuration` property to `DEBUG`.

```
msbuild.exe MyProj.proj -p:Configuration=DEBUG
```

Global properties can also be set or modified for child projects in a multi-project build by using the `Properties` attribute of the MSBuild task. Global properties are also forwarded to child projects unless the `RemoveProperties` attribute of the MSBuild task is used to specify the list of properties not to forward. For more information, see MSBuild task.

If you specify a property by using the `TreatAsLocalProperty` attribute in a project tag, that global property value doesn't override the property value that's set in the project file. For more information, see Project element (MSBuild) and How to: Build the same source files with different options.

Property functions

Starting in .NET Framework version 4, you can use property functions to evaluate your MSBuild scripts. You can read the system time, compare strings, match regular expressions, and perform many other actions within your build script without using MSBuild tasks.

You can use string (instance) methods to operate on any property value, and you can call the static methods of many system classes. For example, you can set a build property to today's date as follows.

```
<Today>$([System.DateTime]::Now.ToString("yyyy.MM.dd"))</Today>
```

For more information, and a list of property functions, see Property functions.

Create properties during execution

Properties positioned outside `Target` elements are assigned values during the evaluation phase of a build. During the subsequent execution phase, properties can be created or modified as follows:

- A property can be emitted by any task. To emit a property, the Task element must have a child Output element that has a `PropertyName` attribute.
- A property can be emitted by the CreateProperty task. This usage is deprecated.
- Starting in the .NET Framework 3.5, `Target` elements may contain `PropertyGroup` elements that may contain property declarations.

Store XML in properties

Properties can contain arbitrary XML, which can help in passing values to tasks or displaying logging information. The following example shows the `ConfigTemplate` property, which has a value that contains XML and other property references. MSBuild replaces the property references by using their respective property values. Property values are assigned in the order in which they appear. Therefore, in this example, `$(MySupportedVersion)`, `$(MyRequiredVersion)`, and `$(MySafeMode)` should have already been defined.

```
<PropertyGroup>
    <ConfigTemplate>
        <Configuration>
            <Startup>
                <SupportedRuntime
                    ImageVersion="$(MySupportedVersion)"
                    Version="$(MySupportedVersion)"/>
                <RequiredRuntime
                    ImageVersion="$(MyRequiredVersion)"
                    Version="$(MyRequiredVersion)"
                    SafeMode="$(MySafeMode)"/>
            </Startup>
        </Configuration>
    </ConfigTemplate>
</PropertyGroup>
```

See also

- MSBuild concepts
- MSBuild
- How to: Use environment variables in a build
- How to: Reference the name or location of the project file
- How to: Build the same source files with different options

- MSBuild reserved and well-known properties
- Property element (MSBuild)

How to: Use environment variables in a build

4/23/2019 • 2 minutes to read • Edit Online

When you build projects, it is often necessary to set build options using information that is not in the project file or the files that comprise your project. This information is typically stored in environment variables.

Reference environment variables

All environment variables are available to the Microsoft Build Engine (MSBuild) project file as properties.

> **NOTE**
> If the project file contains an explicit definition of a property that has the same name as an environment variable, the property in the project file overrides the value of the environment variable.

To use an environment variable in an MSBuild project

- Reference the environment variable the same way you would a variable declared in your project file. For example, the following code references the BIN_PATH environment variable:

  ```
  <FinalOutput>$(BIN_PATH)\MyAssembly.dll</FinalOutput>
  ```

  You can use a `Condition` attribute to provide a default value for a property if the environment variable was not set.

To provide a default value for a property

- Use a `Condition` attribute on a property to set the value only if the property has no value. For example, the following code sets the `ToolsPath` property to *c:\tools* only if the `ToolsPath` environment variable is not set:

  ```
  <ToolsPath Condition="'$(TOOLSPATH)' == ''">c:\tools</ToolsPath>
  ```

  > **NOTE**
  > Property names are not case-sensitive so both `$(ToolsPath)` and `$(TOOLSPATH)` reference the same property or environment variable.

Example

The following project file uses environment variables to specify the location of directories.

```
<Project DefaultTargets="FakeBuild">
    <PropertyGroup>
        <FinalOutput>$(BIN_PATH)\myassembly.dll</FinalOutput>
        <ToolsPath Condition=" '$(ToolsPath)' == '' ">
            C:\Tools
        </ToolsPath>
    </PropertyGroup>
    <Target Name="FakeBuild">
        <Message Text="Building $(FinalOutput) using the tools at $(ToolsPath)..."/>
    </Target>
</Project>
```

See also

- MSBuild
- MSBuild properties
- How to: Build the same source files with different options

How to: Reference the name or location of the project file

4/23/2019 • 2 minutes to read • Edit Online

You can use the name or location of the project in the project file itself without having to create your own property. MSBuild provides reserved properties that reference the project file name and other properties related to the project. For more information on reserved properties, see MSBuild reserved and well-known properties.

Use the project properties

MSBuild provides some reserved properties that you can use in your project files without defining them each time. For example, the reserved property `MSBuildProjectName` provides a reference to the project file name. The reserved property `MSBuildProjectDirectory` provides a reference to the project file location.

To use the project properties

- Reference the property in the project file with the $() notation, just as you would with any property. For example:

```
<CSC Sources = "@(CSFile)"
    OutputAssembly = "$(MSBuildProjectName).exe"/>
</CSC>
```

  An advantage of using a reserved property is that any changes to the project file name are incorporated automatically. The next time that you build the project, the output file will have the new name with no further action required on your part.

> **NOTE**
> Reserved properties cannot be redefined in the project file.

Example

The following example project file references the project name as a reserved property to specify the name for the output.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
    DefaultTargets = "Compile">

    <!-- Specify the inputs -->
    <ItemGroup>
        <CSFile Include = "consolehwcs1.cs"/>
     </ItemGroup>
    <Target Name = "Compile">
        <!-- Run the Visual C# compilation using
        input files of type CSFile -->
        <CSC Sources = "@(CSFile)"
            OutputAssembly = "$(MSBuildProjectName).exe" >
            <!-- Set the OutputAssembly attribute of the CSC task
            to the name of the project -->
            <Output
                TaskParameter = "OutputAssembly"
                ItemName = "EXEFile" />
        </CSC>
        <!-- Log the file name of the output file -->
        <Message Text="The output file is @(EXEFile)"/>
    </Target>
</Project>
```

Example

The following example project file uses the `MSBuildProjectDirectory` reserved property to create the full path to a file in the project file location.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <!-- Build the path to a file in the root of the project -->
    <PropertyGroup>
        <NewFilePath>$([System.IO.Path]::Combine($(MSBuildProjectDirectory), `BuildInfo.txt`))</NewFilePath>
    </PropertyGroup>
</Project>
```

See also

- MSBuild
- MSBuild reserved and well-known properties

How to: Build the same source files with different options

2/21/2019 • 3 minutes to read • Edit Online

When you build projects, you frequently compile the same components with different build options. For example, you can create a debug build with symbol information or a release build with no symbol information but with optimizations enabled. Or you can build a project to run on a specific platform, such as x86 or x64. In all these cases, most of the build options stay the same; only a few options are changed to control the build configuration. With MSBuild, you use properties and conditions to create the different build configurations.

Use properties to modify projects

The `Property` element defines a variable that is referenced several times in a project file, such as the location of a temporary directory, or to set the values for properties that are used in several configurations, such as a Debug build and a Release build. For more information about properties, see MSBuild properties.

You can use properties to change the configuration of your build without having to change the project file. The `Condition` attribute of the `Property` element and the `PropertyGroup` element allows you to change the value of properties. For more information about MSBuild conditions, see Conditions.

To set a group of properties based on another property

- Use a `Condition` attribute in a `PropertyGroup` element similar to the following:

```
<PropertyGroup Condition="'$(Flavor)'=='DEBUG'">
    <DebugType>full</DebugType>
    <Optimize>no</Optimize>
</PropertyGroup>
```

To define a property based on another property

- Use a `Condition` attribute in a `Property` element similar to the following:

```
<DebugType Condition="'$(Flavor)'=='DEBUG'">full</DebugType>
```

Specify properties on the command line

Once your project file is written to accept multiple configurations, you need to have the ability to change those configurations whenever you build your project. MSBuild provides this ability by allowing properties to be specified on the command line using the **-property** or **-p** switch.

To set a project property at the command line

- Use the **-property** switch with the property and property value. For example:

```
msbuild file.proj -property:Flavor=Debug
```

or

```
Msbuild file.proj -p:Flavor=Debug
```

To specify more than one project property at the command line

- Use the **-property** or **-p** switch multiple times with the property and property values, or use one **-property** or **-p** switch and separate multiple properties with semicolons (;). For example:

  ```
  msbuild file.proj -p:Flavor=Debug;Platform=x86
  ```

  or

  ```
  msbuild file.proj -p:Flavor=Debug -p:Platform=x86
  ```

  Environment variables are also treated as properties and are automatically incorporated by MSBuild. For more information about using environment variables, see How to: Use environment variables in a build.

  The property value that is specified on the command line takes precedence over any value that is set for the same property in the project file, and that value in the project file takes precedence over the value in an environment variable.

  You can change this behavior by using the `TreatAsLocalProperty` attribute in a project tag. For property names that are listed with that attribute, the property value that's specified on the command line doesn't take precedence over the value in the project file. You can find an example later in this topic.

Example

The following code example, the "Hello World" project, contains two new property groups that can be used to create a Debug build and a Release build.

To build the debug version of this project, type:

```
msbuild consolehwcs1.proj -p:flavor=debug
```

To build the retail version of this project, type:

```
msbuild consolehwcs1.proj -p:flavor=retail
```

```
<Project DefaultTargets = "Compile"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <!-- Sets the default flavor of an environment variable called
    Flavor is not set or specified on the command line -->
    <PropertyGroup>
        <Flavor Condition="'$(Flavor)'==''">DEBUG</Flavor>
    </PropertyGroup>

    <!-- Define the DEBUG settings -->
    <PropertyGroup Condition="'$(Flavor)'=='DEBUG'">
        <DebugType>full</DebugType>
        <Optimize>no</Optimize>
    </PropertyGroup>

    <!-- Define the RETAIL settings -->
    <PropertyGroup Condition="'$(Flavor)'=='RETAIL'">
        <DebugType>pdbonly</DebugType>
        <Optimize>yes</Optimize>
    </PropertyGroup>

    <!-- Set the application name as a property -->
    <PropertyGroup>
        <appname>HelloWorldCS</appname>
    </PropertyGroup>

    <!-- Specify the inputs by type and file name -->
    <ItemGroup>
        <CSFile Include = "consolehwcs1.cs"/>
    </ItemGroup>

    <Target Name = "Compile">
        <!-- Run the Visual C# compilation using input files
        of type CSFile -->
        <CSC  Sources = "@(CSFile)"
            DebugType="$(DebugType)"
            Optimize="$(Optimize)"
            OutputAssembly="$(appname).exe" >

            <!-- Set the OutputAssembly attribute of the CSC
            task to the name of the executable file that is
            created -->
            <Output TaskParameter="OutputAssembly"
                ItemName = "EXEFile" />
        </CSC>
        <!-- Log the file name of the output file -->
        <Message Text="The output file is @(EXEFile)"/>
    </Target>
</Project>
```

Example

The following example illustrates how to use the `TreatAsLocalProperty` attribute. The `Color` property has a value of `Blue` in the project file and `Green` in the command line. With `TreatAsLocalProperty="Color"` in the project tag, the command-line property (`Green`) doesn't override the property that's defined in the project file (`Blue`).

To build the project, enter the following command:

```
msbuild colortest.proj -t:go -property:Color=Green
```

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
ToolsVersion="4.0" TreatAsLocalProperty="Color">

    <PropertyGroup>
        <Color>Blue</Color>
    </PropertyGroup>

    <Target Name="go">
        <Message Text="Color: $(Color)" />
    </Target>
</Project>

<!--
  Output with TreatAsLocalProperty="Color" in project tag:
     Color: Blue

  Output without TreatAsLocalProperty="Color" in project tag:
     Color: Green
-->
```

See also

- MSBuild
- MSBuild concepts
- MSBuild reference
- Project element (MSBuild)

Property functions

5/15/2019 • 7 minutes to read • Edit Online

In the .NET Framework versions 4 and 4.5, property functions can be used to evaluate MSBuild scripts. Property functions can be used wherever properties appear. Unlike tasks, property functions can be used outside of targets, and are evaluated before any target runs.

Without using MSBuild tasks, you can read the system time, compare strings, match regular expressions, and perform other actions in your build script. MSBuild will try to convert string to number and number to string, and make other conversions as required.

String values returned from property functions have special characters escaped. If you want the value to be treated as though it was put directly in the project file, use `$([MSBuild]::Unescape())` to unescape the special characters.

Property function syntax

These are three kinds of property functions; each function has a different syntax:

- String (instance) property functions
- Static property functions
- MSBuild property functions

String property functions

All build property values are just string values. You can use string (instance) methods to operate on any property value. For example, you can extract the drive name (the first three characters) from a build property that represents a full path by using this code:

```
$(ProjectOutputFolder.Substring(0,3))
```

Static property functions

In your build script, you can access the static properties and methods of many system classes. To get the value of a static property, use the following syntax, where <Class> is the name of the system class and <Property> is the name of the property.

```
$([Class]::Property)
```

For example, you can use the following code to set a build property to the current date and time.

```
<Today>$([System.DateTime]::Now)</Today>
```

To call a static method, use the following syntax, where <Class> is the name of the system class, <Method> is the name of the method, and (<Parameters>) is the parameter list for the method:

```
$([Class]::Method(Parameters))
```

For example, to set a build property to a new GUID, you can use this script:

```
<NewGuid>$([System.Guid]::NewGuid())</NewGuid>
```

In static property functions, you can use any static method or property of these system classes:

- System.Byte
- System.Char
- System.Convert
- System.DateTime
- System.Decimal
- System.Double
- System.Enum
- System.Guid
- System.Int16
- System.Int32
- System.Int64
- System.IO.Path
- System.Math
- System.Runtime.InteropServices.OSPlatform
- System.Runtime.InteropServices.RuntimeInformation
- System.UInt16
- System.UInt32
- System.UInt64
- System.SByte
- System.Single
- System.String
- System.StringComparer
- System.TimeSpan
- System.Text.RegularExpressions.Regex
- System.UriBuilder
- System.Version
- Microsoft.Build.Utilities.ToolLocationHelper

In addition, you can use the following static methods and properties:

- System.Environment::CommandLine
- System.Environment::ExpandEnvironmentVariables
- System.Environment::GetEnvironmentVariable
- System.Environment::GetEnvironmentVariables
- System.Environment::GetFolderPath
- System.Environment::GetLogicalDrives
- System.IO.Directory::GetDirectories
- System.IO.Directory::GetFiles
- System.IO.Directory::GetLastAccessTime
- System.IO.Directory::GetLastWriteTime
- System.IO.Directory::GetParent
- System.IO.File::Exists
- System.IO.File::GetCreationTime
- System.IO.File::GetAttributes

- System.IO.File::GetLastAccessTime
- System.IO.File::GetLastWriteTime
- System.IO.File::ReadAllText

Calling instance methods on static properties

If you access a static property that returns an object instance, you can invoke the instance methods of that object. To invoke an instance method, use the following syntax, where <Class> is the name of the system class, <Property> is the name of the property, <Method> is the name of the method, and (<Parameters>) is the parameter list for the method:

```
$([Class]::Property.Method(Parameters))
```

The name of the class must be fully qualified with the namespace.

For example, you can use the following code to set a build property to the current date today.

```
<Today>$([System.DateTime]::Now.ToString('yyyy.MM.dd'))</Today>
```

MSBuild property functions

Several static methods in your build can be accessed to provide arithmetic, bitwise logical, and escape character support. You access these methods by using the following syntax, where <Method> is the name of the method and (<Parameters>) is the parameter list for the method.

```
$([MSBuild]::Method(Parameters))
```

For example, to add together two properties that have numeric values, use the following code.

```
$([MSBuild]::Add($(NumberOne), $(NumberTwo)))
```

Here is a list of MSBuild property functions:

FUNCTION SIGNATURE	DESCRIPTION
double Add(double a, double b)	Add two doubles.
long Add(long a, long b)	Add two longs.
double Subtract(double a, double b)	Subtract two doubles.
long Subtract(long a, long b)	Subtract two longs.
double Multiply(double a, double b)	Multiply two doubles.
long Multiply(long a, long b)	Multiply two longs.
double Divide(double a, double b)	Divide two doubles.
long Divide(long a, long b)	Divide two longs.
double Modulo(double a, double b)	Modulo two doubles.

FUNCTION SIGNATURE	DESCRIPTION
long Modulo(long a, long b)	Modulo two longs.
string Escape(string unescaped)	Escape the string according to MSBuild escaping rules.
string Unescape(string escaped)	Unescape the string according to MSBuild escaping rules.
int BitwiseOr(int first, int second)	Perform a bitwise `OR` on the first and second (first | second).
int BitwiseAnd(int first, int second)	Perform a bitwise `AND` on the first and second (first & second).
int BitwiseXor(int first, int second)	Perform a bitwise `XOR` on the first and second (first ^ second).
int BitwiseNot(int first)	Perform a bitwise `NOT` (~first).
bool IsOsPlatform(string platformString)	Specify whether the current OS platform is `platformString`. `platformString` must be a member of OSPlatform.
bool IsOSUnixLike()	True if current OS is a Unix system.
string NormalizePath(params string[] path)	Gets the canonicalized full path of the provided path and ensures it contains the correct directory separator characters for the current operating system.
string NormalizeDirectory(params string[] path)	Gets the canonicalized full path of the provided directory and ensures it contains the correct directory separator characters for the current operating system while ensuring it has a trailing slash.
string EnsureTrailingSlash(string path)	If the given path doesn't have a trailing slash then add one. If the path is an empty string, does not modify it.
string GetPathOfFileAbove(string file, string startingDirectory)	Searches for a file based on the current build file's location, or based on `startingDirectory`, if specified.
GetDirectoryNameOfFileAbove(string startingDirectory, string fileName)	Locate a file in either the directory specified or a location in the directory structure above that directory.
string MakeRelative(string basePath, string path)	Makes `path` relative to `basePath`. `basePath` must be an absolute directory. If `path` cannot be made relative, it is returned verbatim. Similar to `Uri.MakeRelativeUri`.
string ValueOrDefault(string conditionValue, string defaultValue)	Return the string in parameter 'defaultValue' only if parameter 'conditionValue' is empty, else, return the value conditionValue.

Nested property functions

You can combine property functions to form more complex functions, as the following example shows.

```
$([MSBuild]::BitwiseAnd(32, $([System.IO.File]::GetAttributes(tempFile))))
```

This example returns the value of the FileAttributes `Archive` bit (32 or 0) of the file given by the path `tempFile`. Notice that enumerated data values cannot appear by name within property functions. The numeric value (32) must be used instead.

Metadata may also appear in nested property functions. For more information, see Batching.

MSBuild DoesTaskHostExist

The `DoesTaskHostExist` property function in MSBuild returns whether a task host is currently installed for the specified runtime and architecture values.

This property function has the following syntax:

```
$([MSBuild]::DoesTaskHostExist(string theRuntime, string theArchitecture))
```

MSBuild EnsureTrailingSlash

The `EnsureTrailingSlash` property function in MSBuild adds a trailing slash if one doesn't already exist.

This property function has the following syntax:

```
$([MSBuild]::EnsureTrailingSlash('$(PathProperty)'))
```

MSBuild GetDirectoryNameOfFileAbove

The MSBuild `GetDirectoryNameOfFileAbove` property function looks for a file in the directories above the current directory in the path.

This property function has the following syntax:

```
$([MSBuild]::GetDirectoryNameOfFileAbove(string ThePath, string TheFile))
```

The following code is an example of this syntax.

```
<Import Project="$([MSBuild]::GetDirectoryNameOfFileAbove($(MSBuildThisFileDirectory),
EnlistmentInfo.props))\EnlistmentInfo.props" Condition="
'$([MSBuild]::GetDirectoryNameOfFileAbove($(MSBuildThisFileDirectory), EnlistmentInfo.props))' != '' " />
```

MSBuild GetPathOfFileAbove

The `GetPathOfFileAbove` property function in MSBuild returns the path of the file immediately preceding this one. It is functionally equivalent to calling

```
<Import Project="$([MSBuild]::GetDirectoryNameOfFileAbove($(MSBuildThisFileDirectory), dir.props))\dir.props"
/>
```

This property function has the following syntax:

```
$([MSBuild]::GetPathOfFileAbove(dir.props))
```

MSBuild GetRegistryValue

The MSBuild `GetRegistryValue` property function returns the value of a registry key. This function takes two arguments, the key name and the value name, and returns the value from the registry. If you don't specify a value name, the default value is returned.

The following examples show how this function is used:

```
$([MSBuild]::GetRegistryValue(`HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\10.0\Debugger`,
``))                                  // default value
$([MSBuild]::GetRegistryValue(`HKEY_CURRENT_USER\Software\Microsoft\VisualStudio\10.0\Debugger`,
`SymbolCacheDir`))
$([MSBuild]::GetRegistryValue(`HKEY_LOCAL_MACHINE\SOFTWARE\(SampleName)`, `(SampleValue)`))             //
parens in name and value
```

MSBuild GetRegistryValueFromView

The MSBuild `GetRegistryValueFromView` property function gets system registry data given the registry key, value, and one or more ordered registry views. The key and value are searched in each registry view in order until they are found.

The syntax for this property function is:

```
[MSBuild]::GetRegistryValueFromView(string keyName, string valueName, object defaultValue, params object[]
views)
```

The Windows 64-bit operating system maintains a **HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node** registry key that presents a **HKEY_LOCAL_MACHINE\SOFTWARE** registry view for 32-bit applications.

By default, a 32-bit application running on WOW64 accesses the 32-bit registry view and a 64-bit application accesses the 64-bit registry view.

The following registry views are available:

REGISTRY VIEW	DEFINITION
RegistryView.Registry32	The 32-bit application registry view.
RegistryView.Registry64	The 64-bit application registry view.
RegistryView.Default	The registry view that matches the process that the application is running on.

The following is an example.

```
$([MSBuild]::GetRegistryValueFromView('HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Microsoft
SDKs\Silverlight\v3.0\ReferenceAssemblies', 'SLRuntimeInstallPath', null, RegistryView.Registry64,
RegistryView.Registry32))
```

gets the **SLRuntimeInstallPath** data of the **ReferenceAssemblies** key, looking first in the 64-bit registry view and then in the 32-bit registry view.

MSBuild MakeRelative

The MSBuild `MakeRelative` property function returns the relative path of the second path relative to first path. Each path can be a file or folder.

This property function has the following syntax:

```
$([MSBuild]::MakeRelative($(FileOrFolderPath1), $(FileOrFolderPath2)))
```

The following code is an example of this syntax.

```
<PropertyGroup>
    <Path1>c:\users\</Path1>
    <Path2>c:\users\username\</Path2>
</PropertyGroup>

<Target Name = "Go">
    <Message Text ="$([MSBuild]::MakeRelative($(Path1), $(Path2)))" />
    <Message Text ="$([MSBuild]::MakeRelative($(Path2), $(Path1)))" />
</Target>

<!--
Output:
   username\
   ..\
-->
```

MSBuild ValueOrDefault

The MSBuild `ValueOrDefault` property function returns the first argument, unless it's null or empty. If the first argument is null or empty, the function returns the second argument.

The following example shows how this function is used.

```
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <PropertyGroup>
        <Value1>$([MSBuild]::ValueOrDefault('$(UndefinedValue)', 'a'))</Value1>
        <Value2>$([MSBuild]::ValueOrDefault('b', '$(Value1)'))</Value2>
    </PropertyGroup>

    <Target Name="MyTarget">
        <Message Text="Value1 = $(Value1)" />
        <Message Text="Value2 = $(Value2)" />
    </Target>
</Project>

<!--
Output:
  Value1 = a
  Value2 = b
-->
```

See also

- MSBuild properties
- MSBuild overview

MSBuild items

9/24/2019 • 8 minutes to read • Edit Online

MSBuild items are inputs into the build system, and they typically represent files (the files are specified in the `Include` attribute). Items are grouped into item types based on their element names. Item types are named lists of items that can be used as parameters for tasks. The tasks use the item values to perform the steps of the build process.

Because items are named by the item type to which they belong, the terms "item" and "item value" can be used interchangeably.

Create items in a project file

You declare items in the project file as child elements of an ItemGroup element. The name of the child element is the type of the item. The `Include` attribute of the element specifies the items (files) to be included with that item type. For example, the following XML creates an item type that's named `Compile`, which includes two files.

```
<ItemGroup>
    <Compile Include = "file1.cs"/>
    <Compile Include = "file2.cs"/>
</ItemGroup>
```

The item *file2.cs* doesn't replace the item *file1.cs*; instead, the file name is appended to the list of values for the `Compile` item type.

The following XML creates the same item type by declaring both files in one `Include` attribute. Notice that the file names are separated by a semicolon.

```
<ItemGroup>
    <Compile Include = "file1.cs;file2.cs"/>
</ItemGroup>
```

Create items during execution

Items that are outside Target elements are assigned values during the evaluation phase of a build. During the subsequent execution phase, items can be created or modified in the following ways:

- Any task can emit an item. To emit an item, the Task element must have a child Output element that has an `ItemName` attribute.
- The CreateItem task can emit an item. This usage is deprecated.
- Starting in the .NET Framework 3.5, `Target` elements may contain ItemGroup elements that may contain item elements.

Reference items in a project file

To reference item types throughout the project file, you use the syntax @(<ItemType>). For example, you would reference the item type in the previous example by using `@(Compile)`. By using this syntax, you can pass items to tasks by specifying the item type as a parameter of that task. For more information, see How to: Select the files to build.

By default, the items of an item type are separated by semicolons (;) when it's expanded. You can use the syntax @(<ItemType>, '<separator>') to specify a separator other than the default. For more information, see How to: Display an item list separated with commas.

Use wildcards to specify items

You can use the `**`, `*`, and `?` wildcard characters to specify a group of files as inputs for a build instead of listing each file separately.

- The `?` wildcard character matches a single character.
- The `*` wildcard character matches zero or more characters.
- The `**` wildcard character sequence matches a partial path.

For example, you can specify all the `.cs` files in the directory that contains the project file by using the following element in your project file.

```
<CSFile Include="*.cs"/>
```

The following element selects all `.vb` files on the `D:` drive:

```
<VBFile Include="D:/**/*.vb"/>
```

If you would like to include literal `*` or `?` characters in an item without wildcard expansion, you must escape the wildcard characters.

For more information about wildcard characters, see How to: Select the files to build.

Use the Exclude attribute

Item elements can contain the `Exclude` attribute, which excludes specific items (files) from the item type. The `Exclude` attribute is typically used together with wildcard characters. For example, the following XML adds every *.cs* file in the directory to the CSFile item type, except the *DoNotBuild.cs* file.

```
<ItemGroup>
    <CSFile  Include="*.cs"  Exclude="DoNotBuild.cs"/>
</ItemGroup>
```

The `Exclude` attribute affects only the items that are added by the `Include` attribute in the item element that contains them both. The following example wouldn't exclude the file *Form1.cs*, which was added in the preceding item element.

```
<Compile Include="*.cs" />
<Compile Include="*.res" Exclude="Form1.cs">
```

For more information, see How to: Exclude files from the build.

Item metadata

Items may contain metadata in addition to the information in the `Include` and `Exclude` attributes. This metadata can be used by tasks that require more information about the items or to batch tasks and targets. For more information, see Batching.

Metadata is a collection of key-value pairs that are declared in the project file as child elements of an item

element. The name of the child element is the name of the metadata, and the value of the child element is the value of the metadata.

The metadata is associated with the item element that contains it. For example, the following XML adds `Culture` metadata that has the value `Fr` to both the *one.cs* and the *two.cs* items of the CSFile item type.

```
<ItemGroup>
    <CSFile Include="one.cs;two.cs">
        <Culture>Fr</Culture>
    </CSFile>
</ItemGroup>
```

An item can have zero or more metadata values. You can change metadata values at any time. If you set metadata to an empty value, you effectively remove it from the build.

Reference item metadata in a project file

You can reference item metadata throughout the project file by using the syntax %(<ItemMetadataName>). If ambiguity exists, you can qualify a reference by using the name of the item type. For example, you can specify %(<ItemType.ItemMetaDataName>).The following example uses the Display metadata to batch the Message task. For more information about how to use item metadata for batching, see Item metadata in task batching.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <ItemGroup>
        <Stuff Include="One.cs" >
            <Display>false</Display>
        </Stuff>
        <Stuff Include="Two.cs">
            <Display>true</Display>
        </Stuff>
    </ItemGroup>
    <Target Name="Batching">
        <Message Text="@(Stuff)" Condition=" '%(Display)' == 'true' "/>
    </Target>
</Project>
```

Well-known item metadata

When an item is added to an item type, that item is assigned some well-known metadata. For example, all items have the well-known metadata %(<Filename>), whose value is the file name of the item. For more information, see Well-known item metadata.

Transform item types by using metadata

You can transform item lists into new item lists by using metadata. For example, you can transform an item type `CppFiles` that has items that represent *.cpp* files into a corresponding list of *.obj* files by using the expression `@(CppFiles -> '%(Filename).obj')`.

The following code creates a `CultureResource` item type that contains copies of all `EmbeddedResource` items with `Culture` metadata. The `Culture` metadata value becomes the value of the new metadata `CultureResource.TargetDirectory`.

```
<Target Name="ProcessCultureResources">
    <ItemGroup>
        <CultureResource Include="@(EmbeddedResource)"
            Condition="'%(EmbeddedResource.Culture)' != ''">
            <TargetDirectory>%(EmbeddedResource.Culture) </TargetDirectory>
        </CultureResource>
    </ItemGroup>
</Target>
```

For more information, see Transforms.

Item definitions

Starting in the .NET Framework 3.5, you can add default metadata to any item type by using the ItemDefinitionGroup element. Like well-known metadata, the default metadata is associated with all items of the item type that you specify. You can explicitly override default metadata in an item definition. For example, the following XML gives the `Compile` items *one.cs* and *three.cs* the metadata `BuildDay` with the value "Monday". The code gives the item *two.cs* the metadata `BuildDay` with the value "Tuesday".

```
<ItemDefinitionGroup>
    <Compile>
        <BuildDay>Monday</BuildDay>
    </Compile>
</ItemDefinitionGroup>
<ItemGroup>
    <Compile Include="one.cs;three.cs" />
    <Compile Include="two.cs">
        <BuildDay>Tuesday</BuildDay>
    </Compile>
</ItemGroup>
```

For more information, see Item definitions.

Attributes for items in an ItemGroup of a Target

Starting in the .NET Framework 3.5, `Target` elements may contain ItemGroup elements that may contain item elements. The attributes in this section are valid when they are specified for an item in an `ItemGroup` that's in a `Target`.

Remove attribute

The `Remove` attribute removes specific items (files) from the item type. This attribute was introduced in the .NET Framework 3.5 (inside targets only). Both inside and outside targets are supported starting in MSBuild 15.0.

The following example removes every *.config* file from the Compile item type.

```
<Target>
    <ItemGroup>
        <Compile Remove="*.config"/>
    </ItemGroup>
</Target>
```

KeepMetadata attribute

If an item is generated within a target, the item element can contain the `KeepMetadata` attribute. If this attribute is specified, only the metadata that is specified in the semicolon-delimited list of names will be transferred from the source item to the target item. An empty value for this attribute is equivalent to not specifying it. The `KeepMetadata` attribute was introduced in the .NET Framework 4.5.

The following example illustrates how to use the `KeepMetadata` attribute.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
ToolsVersion="4.0">

    <ItemGroup>
        <FirstItem Include="rhinoceros">
            <Class>mammal</Class>
            <Size>large</Size>
        </FirstItem>

    </ItemGroup>
    <Target Name="MyTarget">
        <ItemGroup>
            <SecondItem Include="@(FirstItem)" KeepMetadata="Class" />
        </ItemGroup>

        <Message Text="FirstItem: %(FirstItem.Identity)" />
        <Message Text="  Class: %(FirstItem.Class)" />
        <Message Text="  Size:  %(FirstItem.Size)"  />

        <Message Text="SecondItem: %(SecondItem.Identity)" />
        <Message Text="  Class: %(SecondItem.Class)" />
        <Message Text="  Size:  %(SecondItem.Size)"  />
    </Target>
</Project>

<!--
Output:
  FirstItem: rhinoceros
    Class: mammal
    Size:  large
  SecondItem: rhinoceros
    Class: mammal
    Size:
-->
```

RemoveMetadata attribute

If an item is generated within a target, the item element can contain the `RemoveMetadata` attribute. If this attribute is specified, all metadata is transferred from the source item to the target item except metadata whose names are contained in the semicolon-delimited list of names. An empty value for this attribute is equivalent to not specifying it. The `RemoveMetadata` attribute was introduced in the .NET Framework 4.5.

The following example illustrates how to use the `RemoveMetadata` attribute.

```
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <PropertyGroup>
        <MetadataToRemove>Size;Material</MetadataToRemove>
    </PropertyGroup>

    <ItemGroup>
        <Item1 Include="stapler">
            <Size>medium</Size>
            <Color>black</Color>
            <Material>plastic</Material>
        </Item1>
    </ItemGroup>

    <Target Name="MyTarget">
        <ItemGroup>
            <Item2 Include="@(Item1)" RemoveMetadata="$(MetadataToRemove)" />
        </ItemGroup>

        <Message Text="Item1: %(Item1.Identity)" />
        <Message Text="  Size:     %(Item1.Size)" />
        <Message Text="  Color:    %(Item1.Color)" />
        <Message Text="  Material: %(Item1.Material)" />
        <Message Text="Item2: %(Item2.Identity)" />
        <Message Text="  Size:     %(Item2.Size)" />
        <Message Text="  Color:    %(Item2.Color)" />
        <Message Text="  Material: %(Item2.Material)" />
    </Target>
</Project>

<!--
Output:
  Item1: stapler
    Size:     medium
    Color:    black
    Material: plastic
  Item2: stapler
    Size:
    Color:    black
    Material:
-->
```

KeepDuplicates attribute

If an item is generated within a target, the item element can contain the `KeepDuplicates` attribute. `KeepDuplicates` is a `Boolean` attribute that specifies whether an item should be added to the target group if the item is an exact duplicate of an existing item.

If the source and target item have the same Include value but different metadata, the item is added even if `KeepDuplicates` is set to `false`. An empty value for this attribute is equivalent to not specifying it. The `KeepDuplicates` attribute was introduced in the .NET Framework 4.5.

The following example illustrates how to use the `KeepDuplicates` attribute.

```
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <ItemGroup>
        <Item1 Include="hourglass;boomerang" />
        <Item2 Include="hourglass;boomerang" />
    </ItemGroup>

    <Target Name="MyTarget">
        <ItemGroup>
            <Item1 Include="hourglass" KeepDuplicates="false" />
            <Item2 Include="hourglass" />
        </ItemGroup>

        <Message Text="Item1: @(Item1)" />
        <Message Text="  %(Item1.Identity)  Count: @(Item1->Count())" />
        <Message Text="Item2: @(Item2)" />
        <Message Text="  %(Item2.Identity)  Count: @(Item2->Count())" />
    </Target>
</Project>

<!--
Output:
  Item1: hourglass;boomerang
    hourglass  Count: 1
    boomerang  Count: 1
  Item2: hourglass;boomerang;hourglass
    hourglass  Count: 2
    boomerang  Count: 1
-->
```

See also

- Item element (MSBuild)
- Common MSBuild project items
- MSBuild concepts
- MSBuild
- How to: Select the files to build
- How to: Exclude files from the build
- How to: Display an item list separated with commas
- Item definitions
- Batching

How to: Select the files to build

4/23/2019 • 2 minutes to read • Edit Online

When you build a project that contains several files, you can list each file separately in the project file, or you can use wildcards to include all the files in one directory or a nested set of directories.

Specify inputs

Items represent the inputs for a build. For more information on items, see Items.

To include files for a build, they must be included in an item list in the MSBuild project file. Multiple files can be added to item lists by either including the files individually or using wildcards to include many files at once.

To declare items individually

- Use the `Include` attributes similar to following:

  ```
  <CSFile Include="form1.cs"/>
  ```

  or

  ```
  <VBFile Include="form1.vb"/>
  ```

  > **NOTE**
  > If items in an item collection are not in the same directory as the project file, you must specify the full or relative path to the item. For example: `Include="..\..\form2.cs"`.

To declare multiple items

- Use the `Include` attributes similar to following:

  ```
  <CSFile Include="form1.cs;form2.cs"/>
  ```

  or

  ```
  <VBFile Include="form1.vb;form2.vb"/>
  ```

Specify inputs with wildcards

You can also use wildcards to recursively include all files or only specific files from subdirectories as inputs for a build. For more information about wildcards, see Items

The following examples are based on a project that contains graphics files in the following directories and subdirectories, with the project file located in the *Project* directory:

Project\Images\BestJpgs

Project\Images\ImgJpgs

Project\Images\ImgJpgs\Img1

To include all *.jpg* files in the *Images* directory and subdirectories

- Use the following `Include` attribute:

  ```
  Include="Images\**\*.jpg"
  ```

To include all *.jpg* files starting with *img*

- Use the following `Include` attribute:

```
Include="Images\**\img*.jpg"
```

To include all files in directories with names ending in *jpgs*

- Use one of the following `Include` attributes:

```
Include="Images\**\*jpgs\*.*"
```

or

```
Include="Images\**\*jpgs\*"
```

Pass items to a task

In a project file, you can use the @() notation in tasks to specify an entire item list as the input for a build. You can use this notation whether you list all files separately or use wildcards.

To use all Visual C# or Visual Basic files as inputs

- Use the `Include` attributes similar to the following:

```
<CSC Sources="@(CSFile)">...</CSC>
```

or

```
<VBC Sources="@(VBFile)">...</VBC>
```

> **NOTE**
> You must use wildcards with items to specify the inputs for a build; you cannot specify the inputs using the `Sources` attribute in MSBuild tasks such as Csc or Vbc. The following example is not valid in a project file:
>
> ```
> <CSC Sources="*.cs">...</CSC>
> ```

Example

The following code example shows a project that includes all of the input files separately.

```
<Project DefaultTargets="Compile"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003" >
    <PropertyGroup>
        <Builtdir>built</Builtdir>
    </PropertyGroup>

    <ItemGroup>
        <CSFile Include="Form1.cs"/>
        <CSFile Include="AssemblyInfo.cs"/>

        <Reference Include="System.dll"/>
        <Reference Include="System.Data.dll"/>
        <Reference Include="System.Drawing.dll"/>
        <Reference Include="System.Windows.Forms.dll"/>
        <Reference Include="System.XML.dll"/>
    </ItemGroup>

    <Target Name="PreBuild">
        <Exec Command="if not exist $(builtdir) md $(builtdir)"/>
    </Target>

    <Target Name="Compile" DependsOnTargets="PreBuild">
        <Csc Sources="@(CSFile)"
            References="@(Reference)"
            OutputAssembly="$(builtdir)\$(MSBuildProjectName).exe"
            TargetType="exe" />
    </Target>
</Project>
```

Example

The following code example uses a wildcard to include all the *.cs* files.

```
<Project DefaultTargets="Compile"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003" >

    <PropertyGroup>
        <builtdir>built</builtdir>
    </PropertyGroup>

    <ItemGroup>
        <CSFile Include="*.cs"/>

        <Reference Include="System.dll"/>
        <Reference Include="System.Data.dll"/>
        <Reference Include="System.Drawing.dll"/>
        <Reference Include="System.Windows.Forms.dll"/>
        <Reference Include="System.XML.dll"/>
    </ItemGroup>

    <Target Name="PreBuild">
        <Exec Command="if not exist $(builtdir) md $(builtdir)"/>
    </Target>

    <Target Name="Compile" DependsOnTargets="PreBuild">
        <Csc Sources="@(CSFile)"
            References="@(Reference)"
            OutputAssembly="$(builtdir)\$(MSBuildProjectName).exe"
            TargetType="exe" />
    </Target>
</Project>
```

See also

- How to: Exclude files from the build
- Items

How to: Exclude files from the build

4/23/2019 • 2 minutes to read • Edit Online

In a project file you can use wildcards to include all the files in one directory or a nested set of directories as inputs for a build. However, there might be one file in the directory or one directory in a nested set of directories that you do not want to include as input for a build. You can explicitly exclude that file or directory from the list of inputs. There may also be a file in a project that you only want to include under certain conditions. You can explicitly declare the conditions under which a file is included in a build.

Exclude a file or directory from the inputs for a build

Item lists are the input files for a build. The items that you want to include are declared either separately or as a group using the `Include` attribute. For example:

```
<CSFile Include="Form1.cs"/>
<CSFile Include ="File1.cs;File2.cs"/>
<CSFile Include="*.cs"/>
<JPGFile Include="Images\**\*.jpg"/>
```

If you have used wildcards to include all the files in one directory or a nested set of directories as inputs for a build, there might be one or more files in the directory or one directory in the a nested set of directories that you do not want to include. To exclude an item from the item list, use the `Exclude` attribute.

To include all *.cs* or *.vb* files except *Form2*

- Use one of the following `Include` and `Exclude` attributes:

  ```
  <CSFile Include="*.cs" Exclude="Form2.cs"/>
  ```

  or

  ```
  <VBFile Include="*.vb" Exclude="Form2.vb"/>
  ```

To include all *.cs* or *.vb* files except *Form2* and *Form3*

- Use one of the following `Include` and `Exclude` attributes:

  ```
  <CSFile Include="*.cs" Exclude="Form2.cs;Form3.cs"/>
  ```

  or

  ```
  <VBFile Include="*.vb" Exclude="Form2.vb;Form3.vb"/>
  ```

To include all *.jpg* files in subdirectories of the *Images* directory except those in the *Version2* directory

- Use the following `Include` and `Exclude` attributes:

  ```
  <JPGFile
      Include="Images\**\*.jpg"
      Exclude = "Images\**\Version2\*.jpg"/>
  ```

> **NOTE**
>
> You must specify the path for both attributes. If you use an absolute path to specify file locations in the `Include` attribute, you must also use an absolute path in the `Exclude` attribute; if you use a relative path in the `Include` attribute, you must also use a relative path in the `Exclude` attribute.

Use conditions to exclude a file or directory from the inputs for a build

If there are items that you want to include, for example, in a Debug build but not a Release build, you can use the `Condition` attribute to specify the conditions under which to include the item.

To include the file *Formula.vb* only in Release builds

- Use a `Condition` attribute similar to the following:

```
<Compile
    Include="Formula.vb"
    Condition=" '$(Configuration)' == 'Release' " />
```

Example

The following code example builds a project with all of the *.cs* files in the directory except *Form2.cs*.

```
<Project DefaultTargets="Compile"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003" >

    <PropertyGroup>
        <builtdir>built</builtdir>
    </PropertyGroup>

    <ItemGroup>
        <CSFile Include="*.cs" Exclude="Form2.cs"/>

        <Reference Include="System.dll"/>
        <Reference Include="System.Data.dll"/>
        <Reference Include="System.Drawing.dll"/>
        <Reference Include="System.Windows.Forms.dll"/>
        <Reference Include="System.XML.dll"/>
    </ItemGroup>

    <Target Name="PreBuild">
        <Exec Command="if not exist $(builtdir) md $(builtdir)"/>
    </Target>

    <Target Name="Compile" DependsOnTargets="PreBuild">
        <Csc Sources="@(CSFile)"
            References="@(Reference)"
            OutputAssembly="$(builtdir)\$(MSBuildProjectName).exe"
            TargetType="exe" />
    </Target>
</Project>
```

See also

- Items
- MSBuild
- How to: Select the files to build

How to: Display an item list separated with commas

2/21/2019 • 2 minutes to read • Edit Online

When you work with item lists in Microsoft Build Engine (MSBuild), it is sometimes useful to display the contents of those item lists in a way that is easy to read. Or, you might have a task that takes a list of items separated with a special separator string. In both of these cases, you can specify a separator string for an item list.

Separate items in a list with commas

By default, MSBuild uses semicolons to separate items in a list. For example, consider a `Message` element with the following value:

```
<Message Text="This is my list of TXT files: @(TXTFile)"/>
```

When the `@(TXTFile)` item list contains the items *App1.txt*, *App2.txt*, and *App3.txt*, the message is:

```
This is my list of TXT files: App1.txt;App2.txt;App3.txt
```

If you want to change the default behavior, you can specify your own separator. The syntax for specifying an item list separator is:

```
@(ItemListName, '<separator>')
```

The separator can be either a single character or a string and must be enclosed in single quotes.

To insert a comma and a space between items

- Use item notation similar to the following:

  ```
  @(TXTFile, ', ')
  ```

Example

In this example, Exec task runs the findstr tool to find specified text strings in the file, *Phrases.txt*. In the findstr command, literal search strings are indicated by the **-c:** switch, so the item separator, `-c:` is inserted between items in the `@(Phrase)` item list.

For this example, the equivalent command-line command is:

```
findstr /i /c:hello /c:world /c:msbuild phrases.txt
```

```
<Project DefaultTargets = "Find"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003" >

    <ItemGroup>
        <Phrase Include="hello"/>
        <Phrase Include="world"/>
        <Phrase Include="msbuild"/>
    </ItemGroup>

    <Target Name = "Find">
        <!-- Find some strings in a file -->
        <Exec Command="findstr /i /c:@(Phrase, ' /c:') phrases.txt"/>
    </Target>
</Project>
```

See also

- MSBuild reference
- Items

Item definitions

2/21/2019 • 4 minutes to read • Edit Online

MSBuild 2.0 enables the static declaration of items in project files by using the ItemGroup element. However, metadata may be added only at the item level, even if the metadata is identical for all items. Starting in MSBuild 3.5, a project element named ItemDefinitionGroup overcomes this limitation. *ItemDefinitionGroup* lets you define a set of item definitions, which add default metadata values to all items in the named item type.

The *ItemDefinitionGroup* element appears immediately after the Project element of the project file. Item definitions provide the following functionality:

- You can define global default metadata for items outside a target. That is, the same metadata applies to all items of the specified type.
- Item types can have multiple definitions. When additional metadata specifications are added to the type, the last specification takes precedence. (The metadata follows the same import order as properties follow.)
- Metadata can be additive. For example, CDefines values are accumulated conditionally, depending on the properties that are being set. For example, `MT;STD_CALL;DEBUG;UNICODE`.
- Metadata can be removed.
- Conditions can be used to control the inclusion of metadata.

Item metadata default values

Item metadata that is defined in an ItemDefinitionGroup is just a declaration of default metadata. The metadata does not apply unless you define an Item that uses an ItemGroup to contain the metadata values.

> **NOTE**
> In many of the examples in this topic, an ItemDefinitionGroup element is shown but its corresponding ItemGroup definition is omitted for clarity.

Metadata explicitly defined in an ItemGroup takes precedence over metadata in ItemDefinitionGroup. Metadata in ItemDefinitionGroup is applied only for undefined metadata in an ItemGroup. For example:

```
<ItemDefinitionGroup>
    <i>
        <m>m1</m>
        <n>n1</n>
    </i>
</ItemDefinitionGroup>
<ItemGroup>
    <i Include="a">
        <o>o1</o>
        <n>n2</n>
    </i>
</ItemGroup>
```

In this example, the default metadata "m" is applied to Item "i" because metadata "m" is not explicitly defined by Item "i". However, default metadata "n" is not applied to Item "i" because metadata "n" is already defined by Item "i".

> **NOTE**
> XML Element and Parameter names are case-sensitive. Item metadata and Item/Property names are not case-sensitive. Therefore, ItemDefinitionGroup items that have names that differ only by case should be treated as the same ItemGroup.

Value sources

The values for metadata that is defined in an ItemDefinitionGroup can come from many different sources, as follows:

- PropertyGroup Property
- Item from an ItemDefinitionGroup
- Item transform on an ItemDefinitionGroup Item
- Environment variable
- Global property (from the *MSBuild.exe* command line)
- Reserved property
- Well-known metadata on an Item from an ItemDefinitionGroup
- CDATA section <![CDATA[anything here is not parsed]]>

> **NOTE**
> Item metadata from an ItemGroup is not useful in an ItemDefinitionGroup metadata declaration because ItemDefinitionGroup elements are processed before ItemGroup elements.

Additive and multiple definitions

When you add definitions or use multiple ItemDefinitionGroups, remember the following:

- Additional metadata specification is added to the type.
- The last specification takes precedence.

When you have multiple ItemDefinitionGroups, each subsequent specification adds its metadata to the previous definition. For example:

```
<ItemDefinitionGroup>
    <i>
        <m>m1</m>
        <n>n1</n>
    </i>
</ItemDefinitionGroup>
<ItemDefinitionGroup>
    <i>
        <o>o1</o>
    </i>
</ItemDefinitionGroup>
```

In this example, the metadata "o" is added to "m" and "n".

In addition, previously defined metadata values can also be added. For example:

```
<ItemDefinitionGroup>
    <i>
        <m>m1</m>
    </i>
</ItemDefinitionGroup>
<ItemDefinitionGroup>
    <i>
        <m>%(m);m2</m>
    </i>
</ItemDefinitionGroup>
```

In this example, the previously defined value for metadata "m" (m1) is added to the new value (m2), so that the final value is "m1;m2".

> **NOTE**
> This can also occur in the same ItemDefinitionGroup.

When you override the previously defined metadata, the last specification takes precedence. In the following example, the final value of metadata "m" goes from "m1" to "m1a".

```
<ItemDefinitionGroup>
    <i>
        <m>m1</m>
    </i>
</ItemDefinitionGroup>
<ItemDefinitionGroup>
    <i>
        <m>m1a</m>
    </i>
</ItemDefinitionGroup>
```

Use conditions in an ItemDefinitionGroup

You can use conditions in an ItemDefinitionGroup to control the inclusion of metadata. For example:

```
<ItemDefinitionGroup Condition="'$(Configuration)'=='Debug'">
    <i>
        <m>m1</m>
    </i>
</ItemDefinitionGroup>
```

In this case, the default metadata "m1" on item "i" is included only if the value of the "Configuration" property is "Debug".

> **NOTE**
> Only local metadata references are supported in conditions.

References to metadata defined in an earlier ItemDefinitionGroup are local to the item, not the definition group. That is, the scope of the references are item-specific. For example:

```
<ItemDefinitionGroup>
    <test>
        <yes>1</yes>
    </test>
    <i>
        <m>m0</m>
        <m Condition="'%(test.yes)'=='1'">m1</m>
    </i>
</ItemDefinitionGroup>
```

In the above example, item "i" references item "test" in its Condition. This Condition will never be true because MSBuild interprets a reference to another item's metadata in an ItemDefinitionGroup as the empty string. Therefore, "m" would be set to "m0."

```
  <ItemDefinitionGroup>
    <i>
      <m>m0</m>
      <yes>1</yes>
      <m Condition="'%(i.yes)'=='1'">m1</m>
    </i>
  </ItemDefinitionGroup>
```

In the above example, "m" would be set to the value "m1" as the Condition references item "i"'s metadata value for item "yes."

Override and delete metadata

Metadata defined in an ItemDefinitionGroup element can be overridden in a later ItemDefinitionGroup element by setting the metadata value to blank. You can also effectively delete a metadata item by setting it to an empty value. For example:

```
<ItemDefinitionGroup>
    <i>
        <m>m1</m>
    </i>
</ItemDefinitionGroup>
<ItemDefinitionGroup>
    <i>
        <m></m>
    </i>
</ItemDefinitionGroup>
```

The item "i" still contains metadata "m", but its value is now empty.

Scope of metadata

ItemDefinitionGroups have global scope on defined and global properties wherever they are defined. Default metadata definitions in an ItemDefinitionGroup can be self-referential. For example, the following uses a simple metadata reference:

```
<ItemDefinitionGroup>
    <i>
        <m>m1</m>
        <m>%(m);m2</m>
    </i>
</ItemDefinitionGroup>
```

A qualified metadata reference can also be used:

```
<ItemDefinitionGroup>
    <i>
        <m>m1</m>
        <m>%(i.m);m2</m>
    </i>
</ItemDefinitionGroup>
```

However, the following is not valid:

```
<ItemDefinitionGroup>
    <i>
        <m>m1</m>
        <m>@(x)</m>
    </i>
</ItemDefinitionGroup>
```

Beginning in MSBuild 3.5, ItemGroups can also be self-referential. For example:

```
<ItemGroup>
    <item Include="a">
        <m>m1</m>
        <m>%(m);m2</m>
    </item>
</ItemGroup>
```

See also

- Batching

Item functions

2/21/2019 • 2 minutes to read • Edit Online

Starting with MSBuild 4.0, code in tasks and targets can call item functions to get information about the items in the project. These functions simplify getting Distinct() items and are faster than looping through the items.

String item functions

You can use string methods and properties in the .NET Framework to operate on any item value. For String methods, specify the method name. For String properties, specify the property name after "get_".

For items that have multiple strings, the string method or property runs on each string.

The following example shows how to use these string item functions.

```
<ItemGroup>
    <theItem Include="andromeda;tadpole;cartwheel" />
</ItemGroup>

<Target Name = "go">
    <Message Text="IndexOf  @(theItem->IndexOf('r'))" />
    <Message Text="Replace  @(theItem->Replace('tadpole', 'pinwheel'))" />
    <Message Text="Length   @(theItem->get_Length())" />
    <Message Text="Chars    @(theItem->get_Chars(2))" />
</Target>

  <!--
  Output:
    IndexOf  3;-1;2
    Replace  andromeda;pinwheel;cartwheel
    Length   9;7;9
    Chars    d;d;r
  -->
```

Intrinsic item functions

The table below lists the intrinsic functions available for items.

FUNCTION	EXAMPLE	DESCRIPTION
`Count`	`@(MyItem->Count())`	Returns the count of the items.
`DirectoryName`	`@(MyItem->DirectoryName())`	Returns the equivalent of `Path.DirectoryName` for each item.
`Distinct`	`@(MyItem->Distinct())`	Returns items that have distinct `Include` values. Metadata is ignored. The comparison is case insensitive.
`DistinctWithCase`	`@(MyItem->DistinctWithCase())`	Returns items that have distinct `itemspec` values. Metadata is ignored. The comparison is case sensitive.

FUNCTION	EXAMPLE	DESCRIPTION
`Reverse`	`@(MyItem->Reverse())`	Returns the items in reverse order.
`AnyHaveMetadataValue`	`@(MyItem->AnyHaveMetadataValue("MetadataName", "MetadataValue"))`	Returns a `boolean` to indicate whether any item has the given metadata name and value. The comparison is case insensitive.
`ClearMetadata`	`@(MyItem->ClearMetadata())`	Returns items with their metadata cleared. Only the `itemspec` is retained.
`HasMetadata`	`@(MyItem->HasMetadata("MetadataName"))`	Returns items that have the given metadata name. The comparison is case insensitive.
`Metadata`	`@(MyItem->Metadata("MetadataName"))`	Returns the values of the metadata that have the metadata name.
`WithMetadataValue`	`@(MyItem->WithMetadataValue("MetadataName", "MetadataValue"))`	Returns items that have the given metadata name and value. The comparison is case insensitive.

The following example shows how to use intrinsic item functions.

```
<ItemGroup>
    <TheItem Include="first">
        <Plant>geranium</Plant>
    </TheItem>
    <TheItem Include="second">
        <Plant>algae</Plant>
    </TheItem>
    <TheItem Include="third">
        <Plant>geranium</Plant>
    </TheItem>
</ItemGroup>

<Target Name="go">
    <Message Text="MetaData:    @(TheItem->Metadata('Plant'))" />
    <Message Text="HasMetadata: @(theItem->HasMetadata('Plant'))" />
    <Message Text="WithMetadataValue: @(TheItem->WithMetadataValue('Plant', 'geranium'))" />
    <Message Text=" " />
    <Message Text="Count:   @(theItem->Count())" />
    <Message Text="Reverse: @(theItem->Reverse())" />
</Target>

  <!--
  Output:
    MetaData:    geranium;algae;geranium
    HasMetadata: first;second;third
    WithMetadataValue: first;third

    Count:   3
    Reverse: third;second;first
  -->
```

See also

- Items

MSBuild targets

6/14/2019 • 2 minutes to read • Edit Online

Targets group tasks together in a particular order and allow the build process to be factored into smaller units. For example, one target may delete all files in the output directory to prepare for the build, while another compiles the inputs for the project and places them in the empty directory. For more information on tasks, see Tasks.

Declare targets in the project file

Targets are declared in a project file with the Target element. For example, the following XML creates a target named Construct, which then calls the Csc task with the Compile item type.

```
<Target Name="Construct">
    <Csc Sources="@(Compile)" />
</Target>
```

Like MSBuild properties, targets can be redefined. For example,

```
<Target Name="AfterBuild" >
    <Message Text="First occurrence" />
</Target>
<Target Name="AfterBuild" >
    <Message Text="Second occurrence" />
</Target>
```

If AfterBuild executes, it displays only "Second occurrence".

MSBuild is import-order dependent, and the last definition of a target is the definition used.

Target build order

Targets must be ordered if the input to one target depends on the output of another target.

There are several ways to specify the order in which targets run.

- Initial targets
- Default targets
- First target
- Target dependencies
- `BeforeTargets` and `AfterTargets` (MSBuild 4.0)

A target never runs twice during a single build, even if a subsequent target in the build depends on it. Once a target runs, its contribution to the build is complete.

For details and more information about the target build order, see Target build order.

Target batching

A target element may have an `Outputs` attribute which specifies metadata in the form %(<Metadata>). If so,

MSBuild runs the target once for each unique metadata value, grouping or "batching" the items that have that metadata value. For example,

```
<ItemGroup>
    <Reference Include="System.Core">
      <RequiredTargetFramework>3.5</RequiredTargetFramework>
    </Reference>
    <Reference Include="System.Xml.Linq">
      <RequiredTargetFramework>3.5</RequiredTargetFramework>
    </Reference>
    <Reference Include="Microsoft.CSharp">
      <RequiredTargetFramework>4.0</RequiredTargetFramework>
    </Reference>
</ItemGroup>
<Target Name="AfterBuild"
    Outputs="%(Reference.RequiredTargetFramework)">
    <Message Text="Reference:
      @(Reference->'%(RequiredTargetFramework)')" />
</Target>
```

batches the Reference items by their RequiredTargetFramework metadata. The output of the target looks like this:

```
Reference: 3.5;3.5
Reference: 4.0
```

Target batching is seldom used in real builds. Task batching is more common. For more information, see Batching.

Incremental builds

Incremental builds are builds that are optimized so that targets with output files that are up-to-date with respect to their corresponding input files are not executed. A target element can have both `Inputs` and `Outputs` attributes, indicating what items the target expects as input, and what items it produces as output.

If all output items are up-to-date, MSBuild skips the target, which significantly improves the build speed. This is called an incremental build of the target. If only some files are up-to-date, MSBuild executes the target without the up-to-date items. This is called a partial incremental build of the target. For more information, see Incremental builds.

See also

- MSBuild concepts
- How to: Use the same target in multiple project files

Target build order

12/3/2019 • 4 minutes to read • Edit Online

Targets must be ordered if the input to one target depends on the output of another target. You can use these attributes to specify the order in which targets are run:

- `InitialTargets`. This `Project` attribute specifies the targets that will run first, even if targets are specified on the command line or in the `DefaultTargets` attribute.
- `DefaultTargets`. This `Project` attribute specifies which targets are run if a target is not specified explicitly on the command line.
- `DependsOnTargets`. This `Target` attribute specifies targets that must run before this target can run.
- `BeforeTargets` and `AfterTargets`. These `Target` attributes specify that this target should run before or after the specified targets (MSBuild 4.0).

A target is never run twice during a build, even if a subsequent target in the build depends on it. Once a target has been run, its contribution to the build is complete.

Targets may have a `Condition` attribute. If the specified condition evaluates to `false`, the target isn't executed and has no effect on the build. For more information about conditions, see Conditions.

Initial targets

The `InitialTargets` attribute of the Project element specifies targets that will run first, even if targets are specified on the command line or in the `DefaultTargets` attribute. Initial targets are typically used for error checking.

The value of the `InitialTargets` attribute can be a semicolon-delimited, ordered list of targets. The following example specifies that the `Warm` target runs, and then the `Eject` target runs.

```
<Project InitialTargets="Warm;Eject" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
```

Imported projects may have their own `InitialTargets` attributes. All initial targets are aggregated together and run in order.

For more information, see How to: Specify which target to build first.

Default targets

The `DefaultTargets` attribute of the Project element specifies which target or targets are built if a target isn't specified explicitly in a command line.

The value of the `DefaultTargets` attribute can be a semicolon-delimited, ordered list of default targets. The following example specifies that the `Clean` target runs, and then the `Build` target runs.

```
<Project DefaultTargets="Clean;Build" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
```

You can override the default targets by using the **-target** switch on the command line. The following example specifies that the `Build` target runs, and then the `Report` target runs. When you specify targets in this way, any default targets are ignored.

```
msbuild -target:Build;Report
```

If both initial targets and default targets are specified, and if no command-line targets are specified, MSBuild runs the initial targets first, and then runs the default targets.

Imported projects may have their own `DefaultTargets` attributes. The first `DefaultTargets` attribute encountered determines which default targets will run.

For more information, see How to: Specify which target to build first.

First target

If there are no initial targets, default targets, or command-line targets, then MSBuild runs the first target it encounters in the project file or any imported project files.

Target dependencies

Targets can describe dependency relationships with each other. The `DependsOnTargets` attribute indicates that a target depends on other targets. For example,

```
<Target Name="Serve" DependsOnTargets="Chop;Cook" />
```

tells MSBuild that the `Serve` target depends on the `Chop` target and the `Cook` target. MSBuild runs the `Chop` target, and then runs the `Cook` target before it runs the `Serve` target.

BeforeTargets and AfterTargets

In MSBuild 4.0, you can specify target order by using the `BeforeTargets` and `AfterTargets` attributes.

Consider the following script.

```
<Project DefaultTargets="Compile;Link" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <Target Name="Compile">
        <Message Text="Compiling" />
    </Target>
    <Target Name="Link">
        <Message Text="Linking" />
    </Target>
</Project>
```

To create an intermediate target `Optimize` that runs after the `Compile` target, but before the `Link` target, add the following target anywhere in the `Project` element.

```
<Target Name="Optimize"
    AfterTargets="Compile" BeforeTargets="Link">
    <Message Text="Optimizing" />
</Target>
```

Determine the target build order

MSBuild determines the target build order as follows:

1. `InitialTargets` targets are run.
2. Targets specified on the command line by the **-target** switch are run. If you specify no targets on the command line, then the `DefaultTargets` targets are run. If neither is present, then the first target

encountered is run.

3. The `Condition` attribute of the target is evaluated. If the `Condition` attribute is present and evaluates to `false`, the target isn't executed and has no further effect on the build.

   Other targets that list the conditional target in `BeforeTargets` or `AfterTargets` still execute in the prescribed order.

4. Before the target is executed or skipped, its `DependsOnTargets` targets are run, unless the `Condition` attribute is applied to the target and evaluates to `false`.

   > **NOTE**
   > A target is considered skipped if it is not executed because its output items are up-to-date (see incremental build). This check is done just before executing the tasks inside target, and does not affect the order of execution of targets.

5. Before the target is executed or skipped, any other target that lists the target in a `BeforeTargets` attribute is run.

6. Before the target is executed, its `Inputs` attribute and `Outputs` attribute are compared. If MSBuild determines that any output files are out of date with respect to the corresponding input file or files, then MSBuild executes the target. Otherwise, MSBuild skips the target.

7. After the target is executed or skipped, any other target that lists it in an `AfterTargets` attribute is run.

See also

- Targets

Incremental builds

4/15/2019 • 3 minutes to read • Edit Online

Incremental builds are builds that are optimized so that targets that have output files that are up-to-date with respect to their corresponding input files are not executed. A target element can have both an `Inputs` attribute, which indicates what items the target expects as input, and an `Outputs` attribute, which indicates what items it produces as output. MSBuild attempts to find a 1-to-1 mapping between the values of these attributes. If a 1-to-1 mapping exists, MSBuild compares the time stamp of every input item to the time stamp of its corresponding output item. Output files that have no 1-to-1 mapping are compared to all input files. An item is considered up-to-date if its output file is the same age or newer than its input file or files.

If all output items are up-to-date, MSBuild skips the target. This *incremental build* of the target can significantly improve the build speed. If only some files are up-to-date, MSBuild executes the target but skips the up-to-date items, and thereby brings all items up-to-date. This process is known as a *partial incremental build*.

1-to-1 mappings are typically produced by item transformations. For more information, see Transforms.

Consider the following target.

```
<Target Name="Backup" Inputs="@(Compile)"
    Outputs="@(Compile->'$(BackupFolder)%(Identity).bak')">
    <Copy SourceFiles="@(Compile)" DestinationFiles=
        "@(Compile->'$(BackupFolder)%(Identity).bak')" />
</Target>
```

The set of files represented by the `Compile` item type is copied to a backup directory. The backup files have the *.bak* file name extension. If the files represented by the `Compile` item type, or the corresponding backup files, are not deleted or modified after the Backup target is run, then the Backup target is skipped in subsequent builds.

Output inference

MSBuild compares the `Inputs` and `Outputs` attributes of a target to determine whether the target has to execute. Ideally, the set of files that exists after an incremental build is completed should remain the same whether or not the associated targets are executed. Because properties and items that are created or altered by tasks can affect the build, MSBuild must infer their values even if the target that affects them is skipped. This process is known as *output inference*.

There are three cases:

- The target has a `Condition` attribute that evaluates to `false`. In this case, the target is not run, and has no effect on the build.
- The target has out-of-date outputs and is run to bring them up-to-date.
- The target has no out-of-date outputs and is skipped. MSBuild evaluates the target and makes changes to items and properties as if the target had been run.

To support incremental compilation, tasks must ensure that the `TaskParameter` attribute value of any `Output` element is equal to a task input parameter. Here are some examples:

```
<CreateProperty Value="123">
    <Output PropertyName="Easy" TaskParameter="Value" />
</CreateProperty>
```

This code creates the property Easy, which has the value "123" whether or not the target is executed or skipped.

Starting in MSBuild 3.5, output inference is performed automatically on item and property groups in a target. `CreateItem` tasks are not required in a target and should be avoided. Also, `CreateProperty` tasks should be used in a target only to determine whether a target has been executed.

Prior to MSBuild 3.5, you can use the CreateItem task.

Determine whether a target has been run

Because of output inference, you have to add a `CreateProperty` task to a target to examine properties and items so that you can determine whether the target has been executed. Add the `CreateProperty` task to the target and give it an `Output` element whose `TaskParameter` is "ValueSetByTask".

```
<CreateProperty Value="true">
    <Output TaskParameter="ValueSetByTask" PropertyName="CompileRan" />
</CreateProperty>
```

This code creates the property CompileRan and gives it the value `true`, but only if the target is executed. If the target is skipped, CompileRan is not created.

See also

- Targets

How to: Specify which target to build first

11/22/2019 • 2 minutes to read • Edit Online

A project file can contain one or more `Target` elements that define how the project is built. The Microsoft Build Engine (MSBuild) engine builds the first project it finds, and any dependencies, unless the project file contains a `DefaultTargets` attribute, an `InitialTargets` attribute, or a target is specified at the command line using the **-target** switch.

Use the InitialTargets attribute

The `InitialTargets` attribute of the `Project` element specifies a target that will run first, even if targets are specified on the command line or in the `DefaultTargets` attribute.

To specify one initial target

- Specify the default target in the `InitialTargets` attribute of the `Project` element. For example:

  ```
  <Project InitialTargets="Clean">
  ```

  You can specify more than one initial target in the `InitialTargets` attribute by listing the targets in order, and using a semicolon to separate each target. The targets in the list will be run sequentially.

To specify more than one initial target

- List the initial targets, separated by semicolons, in the `InitialTargets` attribute of the `Project` element. For example, to run the `Clean` target and then the `Compile` target, type:

  ```
  <Project InitialTargets="Clean;Compile">
  ```

Use the DefaultTargets attribute

The `DefaultTargets` attribute of the `Project` element specifies which target or targets are built if a target is not specified explicitly on the command line. If targets are specified in both the `InitialTargets` and `DefaultTargets` attributes and no target is specified on the command line, MSBuild runs the targets specified in the `InitialTargets` attribute followed by the targets specified in the `DefaultTargets` attribute.

To specify one default target

- Specify the default target in the `DefaultTargets` attribute of the `Project` element. For example:

  ```
  <Project DefaultTargets="Compile">
  ```

  You can specify more than one default target in the `DefaultTargets` attribute by listing the targets in order, and using a semicolon to separate each target. The targets in the list will be run sequentially.

To specify more than one default target

- List the default targets, separated by semicolons, in the `DefaultTargets` attribute of the `Project` element. For example, to run the `Clean` target and then the `Compile` target, type:

  ```
  <Project DefaultTargets="Clean;Compile">
  ```

Use the -target Switch

If a default target is not defined in the project file, or if you do not want to use that default target, you can use the command line switch **-target** to specify a different target. The target or targets specified with the **-target** switch are run instead of the targets specified by the `DefaultTargets` attribute. Targets specified in the `InitialTargets`

attribute always run first.

To use a target other than the default target first

- Specify the target as the first target using the **-target** command line switch. For example:

```
msbuild file.proj -target:Clean
```

To use several targets other than the default targets first

- List the targets, separated by semicolons or commas, using the **-target** command line switch. For example:

```
msbuild <file name>.proj -t:Clean;Compile
```

See also

- MSBuild
- Targets
- How to: Clean a build

How to: Use the same target in multiple project files

4/18/2019 • 2 minutes to read • Edit Online

If you have authored several MSBuild project files, you might have discovered that you need to use the same tasks and targets in different project files. Instead of including the complete description of those tasks or targets in every project file, you can save a target in a separate project file and then import that project into any other project that needs to use the target.

Use the Import element

The `Import` element is used to insert one project file into another project file. The project file that is being imported must be a valid MSBuild project file and contain well-formed XML. The `Project` attribute specifies the path to the imported project file. For more information on the `Import` element, see Import element (MSBuild).

To import a project

1. Define, in the importing project file, all properties and items that are used as parameters for properties and items in the imported project.
2. Use the `Import` element to import the project. For example:

   ```
   <Import Project="MyCommon.targets"/>
   ```

3. Following the `Import` element, define all properties and items that must override default definitions of properties and items in the imported project.

Order of evaluation

When MSBuild reaches an `Import` element, the imported project is effectively inserted into the importing project at the location of the `Import` element. Therefore, the location of the `Import` element can affect the values of properties and items. It is important to understand the properties and items that are set by the imported project, and the properties and items that the imported project uses.

When the project builds, all properties are evaluated first, followed by items. For example, the following XML defines the imported project file *MyCommon.targets*:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <PropertyGroup>
        <Name>MyCommon</Name>
    </PropertyGroup>

    <Target Name="Go">
        <Message Text="Name=$(Name)"/>
    </Target>
</Project>
```

The following XML defines *MyApp.proj*, which imports *MyCommon.targets*:

```
<Project
    DefaultTargets="Go"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <PropertyGroup>
        <Name>MyApp</Name>
    </PropertyGroup>
    <Import Project="MyCommon.targets"/>
</Project>
```

When the project builds, the following message is displayed:

```
Name="MyCommon"
```

Because the project is imported after the property `Name` has been defined in *MyApp.proj*, the definition of `Name` in *MyCommon.targets* overrides the definition in *MyApp.proj*. If the project is imported before the property Name is defined, the build would display the following message:

```
Name="MyApp"
```

Use the following approach when importing projects

1. Define, in the project file, all properties and items that are used as parameters for properties and items in the imported project.
2. Import the project.
3. Define in the project file all properties and items that must override default definitions of properties and items in the imported project.

Example

The following code example shows the *MyCommon.targets* file that the second code example imports. The *.targets* file evaluates properties from the importing project to configure the build.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <PropertyGroup>
        <Flavor Condition="'$(Flavor)'==''">DEBUG</Flavor>
        <Optimize Condition="'$(Flavor)'=='RETAIL'">yes</Optimize>
        <appname>$(MSBuildProjectName)</appname>
    <PropertyGroup>
    <Target Name="Build">
        <Csc Sources="hello.cs"
            Optimize="$(Optimize)"
            OutputAssembly="$(appname).exe"/>
    </Target>
</Project>
```

Example

The following code example imports the *MyCommon.targets* file.

```
<Project DefaultTargets="Build"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <PropertyGroup>
        <Flavor>RETAIL</Flavor>
    </PropertyGroup>
    <Import Project="MyCommon.targets"/>
</Project>
```

See also

- Import element (MSBuild)
- Targets

How to: Build specific targets in solutions by using MSBuild.exe

5/28/2019 • 2 minutes to read • Edit Online

You can use *MSBuild.exe* to build specific targets of specific projects in a solution.

To build a specific target of a specific project in a solution

1. At the command line, type `MSBuild.exe <SolutionName>.sln`, where `<SolutionName>` corresponds to the file name of the solution that contains the target that you want to execute.
2. Specify the target after the `-target:` switch in the format <ProjectName>:<TargetName>. If the project name contains any of the characters `%`, `$`, `@`, `;`, `.`, `(`, `)`, or `'`, replace them with an `_` in the specified target name.

Example

The following example executes the `Rebuild` target of the `NotInSlnFolder` project, and then executes the `Clean` target of the `InSolutionFolder` project, which is located in the *NewFolder* solution folder.

```
msbuild SlnFolders.sln -target:NotInSlnfolder:Rebuild;NewFolder\InSolutionFolder:Clean
```

Troubleshooting

If you would like to examine the options available to you, you can use a debugging option provided by MSBuild to do so. Set the environment variable `MSBUILDEMITSOLUTION=1` and build your solution. This will produce an MSBuild file named *<SolutionName>.sln.metaproj* that shows MSBuild's internal view of the solution at build time. You can inspect this view to determine what targets are available to build.

Do not build with this environment variable set unless you need this internal view. This setting can cause problems building projects in your solution.

See also

- Command-line reference
- MSBuild reference
- MSBuild
- MSBuild concepts

How to: Build incrementally

11/22/2019 • 3 minutes to read • Edit Online

When you build a large project, it is important that previously built components that are still up-to-date are not rebuilt. If all targets are built every time, each build will take a long time to complete. To enable incremental builds (builds in which only those targets that have not been built before or targets that are out of date, are rebuilt), the Microsoft Build Engine (MSBuild) can compare the timestamps of the input files with the timestamps of the output files and determine whether to skip, build, or partially rebuild a target. However, there must be a one-to-one mapping between inputs and outputs. You can use transforms to enable targets to identify this direct mapping. For more information on transforms, see Transforms.

Specify inputs and outputs

A target can be built incrementally if the inputs and outputs are specified in the project file.

To specify inputs and outputs for a target

- Use the `Inputs` and `Outputs` attributes of the `Target` element. For example:

```
<Target Name="Build"
    Inputs="@(CSFile)"
    Outputs="hello.exe">
```

MSBuild can compare the timestamps of the input files with the timestamps of the output files and determine whether to skip, build, or partially rebuild a target. In the following example, if any file in the `@(CSFile)` item list is newer than the *hello.exe* file, MSBuild will run the target; otherwise it will be skipped:

```
<Target Name="Build"
    Inputs="@(CSFile)"
    Outputs="hello.exe">

    <Csc
        Sources="@(CSFile)"
        OutputAssembly="hello.exe"/>
</Target>
```

When inputs and outputs are specified in a target, either each output can map to only one input or there can be no direct mapping between the outputs and inputs. In the previous Csc task, for example, the output, *hello.exe*, cannot be mapped to any single input - it depends on all of them.

> **NOTE**
> A target in which there is no direct mapping between the inputs and outputs will always build more often than a target in which each output can map to only one input because MSBuild cannot determine which outputs need to be rebuilt if some of the inputs have changed.

Tasks in which you can identify a direct mapping between the outputs and inputs, such as the LC task, are most suitable for incremental builds, unlike tasks such as Csc and Vbc, which produce one output assembly from a number of inputs.

Example

The following example uses a project that builds Help files for a hypothetical Help system. The project works by converting source *.txt* files into intermediate *.content* files, which then are combined with XML metadata files to produce the final *.help* file used by the Help system. The project uses the following hypothetical tasks:

- `GenerateContentFiles`: Converts *.txt* files into *.content* files.
- `BuildHelp`: Combines *.content* files and XML metadata files to build the final *.help* file.

The project uses transforms to create a one-to-one mapping between inputs and outputs in the `GenerateContentFiles` task. For more information, see Transforms. Also, the `Output` element is set to automatically use the outputs from the `GenerateContentFiles` task as the inputs for the `BuildHelp` task.

This project file contains both the `Convert` and `Build` targets. The `GenerateContentFiles` and `BuildHelp` tasks are placed in the `Convert` and `Build` targets respectively so that each target can be built incrementally. By using the `Output` element, the outputs of the `GenerateContentFiles` task are placed in the `ContentFile` item list, where they can be used as inputs for the `BuildHelp` task. Using the `Output` element in this way automatically provides the outputs from one task as the inputs for another task so that you do not have to list the individual items or item lists manually in each task.

> **NOTE**
> Although the `GenerateContentFiles` target can build incrementally, all outputs from that target always are required as inputs for the `BuildHelp` target. MSBuild automatically provides all the outputs from one target as inputs for another target when you use the `Output` element.

```
<Project DefaultTargets="Build"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003" >

    <ItemGroup>
        <TXTFile Include="*.txt"/>
        <XMLFiles Include="\metadata\*.xml"/>
    </ItemGroup>

    <Target Name = "Convert"
        Inputs="@(TXTFile)"
        Outputs="@(TXTFile->'%(Filename).content')">

        <GenerateContentFiles
            Sources = "@(TXTFile)">
            <Output TaskParameter = "OutputContentFiles"
                ItemName = "ContentFiles"/>
        </GenerateContentFiles>
    </Target>

    <Target Name = "Build" DependsOnTargets = "Convert"
        Inputs="@(ContentFiles);@(XMLFiles)"
        Outputs="$(MSBuildProjectName).help">

        <BuildHelp
            ContentFiles = "@(ContentFiles)"
            MetadataFiles = "@(XMLFiles)"
            OutputFileName = "$(MSBuildProjectName).help"/>
    </Target>
</Project>
```

See also

- Targets
- Target element (MSBuild)

- Transforms
- Csc task
- Vbc task

How to: Clean a build

4/18/2019 • 2 minutes to read • Edit Online

When you clean a build, all intermediate and output files are deleted, leaving only the project and component files. From the project and component files, new instances of the intermediate and output files can then be built. The library of common tasks that is provided with MSBuild includes an Exec task that you can use to run system commands. For more information on the library of tasks, see Task reference.

Create a directory for output items

By default, the *.exe* file that is created when you compile a project is placed in the same directory as the project and source files. Typically, however, output items are created in a separate directory.

To create a directory for output items

1. Use the `Property` element to define the location and name of the directory. For example, create a directory named *BuiltApp* in the directory that contains the project and source files:

   ```
   <builtdir>BuiltApp</builtdir>
   ```

2. Use the MakeDir task to create the directory if the directory does not exist. For example:

   ```
   <MakeDir Directories = "$(builtdir)"
    Condition = "!Exists('$(builtdir)')" />
   ```

Remove the output items

Prior to creating new instances of intermediate and output files, you may want to clear all previous instances of intermediate and output files. Use the RemoveDir task to delete a directory and all files and directories that it contains from a disk.

To remove a directory and all files contained in the directory

- Use the `RemoveDir` task to remove the directory. For example:

  ```
  <RemoveDir Directories="$(builtdir)" />
  ```

Example

The following code example project contains a new target, `Clean`, that uses the `RemoveDir` task to delete a directory and all files and directories that it contains. Also in this example, the `Compile` target creates a separate directory for the output items that are deleted when the build is cleaned.

`Compile` is defined as the default target and is therefore used automatically unless you specify a different target or targets. You use the command line switch **-target** to specify a different target. For example:

```
msbuild <file name>.proj -target:Clean
```

The **-target** switch can be shortened to **-t** and can specify more than one target. For example, to use the target `Clean` then the target `Compile`, type:

```
msbuild <file name>.proj -t:Clean;Compile
```

```
<Project DefaultTargets = "Compile"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003" >

    <PropertyGroup>
        <!-- Set the application name as a property -->
        <name>HelloWorldCS</name>

        <!-- Set the output folder as a property -->
        <builtdir>BuiltApp</builtdir>
    </PropertyGroup>

    <ItemGroup>
        <!-- Specify the inputs by type and file name -->
        <CSFile Include = "consolehwcs1.cs"/>
    </ItemGroup>

    <Target Name = "Compile">
        <!-- Check whether an output folder exists and create
        one if necessary -->
        <MakeDir Directories = "$(builtdir)"
            Condition = "!Exists('$(builtdir)')" />

        <!-- Run the Visual C# compiler -->
        <CSC Sources = "@(CSFile)"
            OutputAssembly = "$(BuiltDir)\$(appname).exe">
            <Output TaskParameter = "OutputAssembly"
                ItemName = "EXEFile" />
        </CSC>

        <!-- Log the file name of the output file -->
        <Message Text="The output file is @(EXEFile)"/>
    </Target>

    <Target Name = "Clean">
        <RemoveDir Directories="$(builtdir)" />
    </Target>
</Project>
```

See also

- Exec task
- MakeDir task
- RemoveDir task
- Csc task
- Targets

How to: Use MSBuild project SDKs

10/24/2019 • 2 minutes to read • Edit Online

MSBuild 15.0 introduced the concept of the "project SDK", which simplifies using software development kits that require properties and targets to be imported.

```
<Project Sdk="Microsoft.NET.Sdk">
    <PropertyGroup>
        <TargetFramework>net46</TargetFramework>
    </PropertyGroup>
</Project>
```

During evaluation of the project, MSBuild adds implicit imports at the top and bottom of your project:

```
<Project>
    <!-- Implicit top import -->
    <Import Project="Sdk.props" Sdk="Microsoft.NET.Sdk" />

    <PropertyGroup>
        <TargetFramework>net46</TargetFramework>
    </PropertyGroup>

    <!-- Implicit bottom import -->
    <Import Project="Sdk.targets" Sdk="Microsoft.NET.Sdk" />
</Project>
```

Reference a project SDK

There are three ways to reference a project SDK:

1. Use the `Sdk` attribute on the `<Project/>` element:

   ```
   <Project Sdk="My.Custom.Sdk">
       ...
   </Project>
   ```

   An implicit import is added to the top and bottom of the project as discussed above.

   To specify a specific version of the SDK you may append it to the `Sdk` attribute:

   ```
   <Project Sdk="My.Custom.Sdk/1.2.3">
       ...
   </Project>
   ```

   > **NOTE**
   > This is currently the only supported way to reference a project SDK in Visual Studio for Mac.

2. Use the top-level `<Sdk/>` element:

```
<Project>
    <Sdk Name="My.Custom.Sdk" Version="1.2.3" />
    ...
</Project>
```

An implicit import is added to the top and bottom of the project as discussed above. The `Version` attribute is not required.

3. Use the `<Import/>` element anywhere in your project:

```
<Project>
    <PropertyGroup>
        <MyProperty>Value</MyProperty>
    </PropertyGroup>
    <Import Project="Sdk.props" Sdk="My.Custom.Sdk" />
    ...
    <Import Project="Sdk.targets" Sdk="My.Custom.Sdk" />
</Project>
```

Explicitly including the imports in your project allows you full control over the order.

When using the `<Import/>` element, you can specify an optional `Version` attribute as well. For example, you can specify `<Import Project="Sdk.props" Sdk="My.Custom.Sdk" Version="1.2.3" />`.

How project SDKs are resolved

When evaluating the import, MSBuild dynamically resolves the path to the project SDK based on the name and version you specified. MSBuild also has a list of registered SDK resolvers which are plug-ins that locate project SDKs on your machine. These plug-ins include:

1. A NuGet-based resolver that queries your configured package feeds for NuGet packages that match the ID and version of the SDK you specified.
   This resolver is only active if you specified an optional version and it can be used for any custom project SDK.
2. A .NET CLI resolver that resolves SDKs that are installed with .NET CLI.
   This resolver locates project SDKs such as `Microsoft.NET.Sdk` and `Microsoft.NET.Sdk.Web` which are part of the product.
3. A default resolver that resolves SDKs that were installed with MSBuild.

The NuGet-based SDK resolver supports specifying a version in your global.json that allows you to control the project SDK version in one place rather than in each individual project:

```
{
    "msbuild-sdks": {
        "My.Custom.Sdk": "5.0.0",
        "My.Other.Sdk": "1.0.0-beta"
    }
}
```

Only one version of each project SDK can be used during a build. If you are referencing two different versions of the same project SDK, MSBuild will emit a warning. It is recommended to **not** specify a version in your projects if a version is specified in your *global.json*.

See also

- MSBuild concepts

- Customize your build
- Packages, metadata, and frameworks
- Additions to the csproj format for .NET Core

MSBuild tasks

2/21/2019 • 2 minutes to read • Edit Online

A build platform needs the ability to execute any number of actions during the build process. MSBuild uses *tasks* to perform these actions. A task is a unit of executable code used by MSBuild to perform atomic build operations.

Task logic

The MSBuild XML project file format cannot fully execute build operations on its own, so task logic must be implemented outside of the project file.

The execution logic of a task is implemented as a .NET class that implements the ITask interface, which is defined in the Microsoft.Build.Framework namespace.

The task class also defines the input and output parameters available to the task in the project file. All public settable non-static non-abstract properties exposed by the task class can be accessed in the project file by placing a corresponding attribute with the same name on the Task element.

You can write your own task by authoring a managed class that implements the ITask interface. For more information, see Task writing.

Execute a task from a project file

Before executing a task in your project file, you must first map the type in the assembly that implements the task to the task name with the UsingTask element. This lets MSBuild know where to look for the execution logic of your task when it finds it in your project file.

To execute a task in an MSBuild project file, create an element with the name of the task as a child of a `Target` element. If a task accepts parameters, these are passed as attributes of the element.

MSBuild item lists and properties can be used as parameters. For example, the following code calls the `MakeDir` task and sets the value of the `Directories` property of the `MakeDir` object equal to the value of the `BuildDir` property declared in the previous example.

```
<Target Name="MakeBuildDirectory">
    <MakeDir
        Directories="$(BuildDir)" />
</Target>
```

Tasks can also return information to the project file, which can be stored in items or properties for later use. For example, the following code calls the `Copy` task and stores the information from the `CopiedFiles` output property in the `SuccessfullyCopiedFiles` item list.

```
<Target Name="CopyFiles">
    <Copy
        SourceFiles="@(MySourceFiles)"
        DestinationFolder="@(MyDestFolder)">
        <Output
            TaskParameter="CopiedFiles"
            ItemName="SuccessfullyCopiedFiles"/>
     </Copy>
</Target>
```

Included tasks

MSBuild ships with many tasks such as Copy, which copies files, MakeDir, which creates directories, and Csc, which compiles Visual C# source code files. For a complete list of available tasks and usage information, see Task reference.

Overridden tasks

MSBuild looks for tasks in several locations. The first location is in files with the extension *.OverrideTasks* stored in the .NET Framework directories. Tasks in these files override any other tasks with the same names, including tasks in the project file. The second location is in files with the extension *.Tasks* in the .NET Framework directories. If the task is not found in either of these locations, the task in the project file is used.

See also

- MSBuild concepts
- MSBuild
- Task writing
- Inline tasks

Task writing

9/24/2019 • 5 minutes to read • Edit Online

Tasks provide the code that runs during the build process. Tasks are contained in targets. A library of typical tasks is included with MSBuild, and you can also create your own tasks. For more information about the library of tasks that are included with MSBuild, see Task reference.

Tasks

Examples of tasks include Copy, which copies one or more files, MakeDir, which creates a directory, and Csc, which compiles Visual C# source code files. Each task is implemented as a .NET class that implements the ITask interface, which is defined in the *Microsoft.Build.Framework.dll* assembly.

There are two approaches you can use when implementing a task:

- Implement the ITask interface directly.
- Derive your class from the helper class, Task, which is defined in the *Microsoft.Build.Utilities.dll* assembly. Task implements ITask and provides default implementations of some ITask members. Additionally, logging is easier.

In both cases, you must add to your class a method named `Execute`, which is the method that is called when the task runs. This method takes no parameters and returns a `Boolean` value: `true` if the task succeeded or `false` if it failed. The following example shows a task that performs no action and returns `true`.

```
using System;
using Microsoft.Build.Framework;
using Microsoft.Build.Utilities;

namespace MyTasks
{
    public class SimpleTask : Task
    {
        public override bool Execute()
        {
            return true;
        }
    }
}
```

The following project file runs this task:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <Target Name="MyTarget">
        <SimpleTask />
    </Target>
</Project>
```

When tasks run, they can also receive inputs from the project file if you create .NET properties on the task class. MSBuild sets these properties immediately before calling the task's `Execute` method. To create a string property, use task code such as:

```
using System;
using Microsoft.Build.Framework;
using Microsoft.Build.Utilities;

namespace MyTasks
{
    public class SimpleTask : Task
    {
        public override bool Execute()
        {
            return true;
        }

        public string MyProperty { get; set; }
    }
}
```

The following project file runs this task and sets `MyProperty` to the given value:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
   <Target Name="MyTarget">
      <SimpleTask MyProperty="Value for MyProperty" />
   </Target>
</Project>
```

Register tasks

If a project is going to run a task, MSBuild must know how to locate the assembly that contains the task class. Tasks are registered using the UsingTask element (MSBuild).

The MSBuild file *Microsoft.Common.Tasks* is a project file that contains a list of `UsingTask` elements that register all the tasks that are supplied with MSBuild. This file is automatically included when building every project. If a task that is registered in *Microsoft.Common.Tasks* is also registered in the current project file, the current project file takes precedence; that is, you can override a default task with your own task that has the same name.

> **TIP**
> You can see a list of the tasks that are supplied with MSBuild by viewing the contents of *Microsoft.Common.Tasks*.

Raise events from a task

If your task derives from the Task helper class, you can use any of the following helper methods on the Task class to raise events that will be caught and displayed by any registered loggers:

```
public override bool Execute()
{
    Log.LogError("messageResource1", "1", "2", "3");
    Log.LogWarning("messageResource2");
    Log.LogMessage(MessageImportance.High, "messageResource3");
    ...
}
```

If your task implements ITask directly, you can still raise such events but you must use the IBuildEngine interface. The following example shows a task that implements ITask and raises a custom event:

```
public class SimpleTask : ITask
{
    public IBuildEngine BuildEngine { get; set; }

    public override bool Execute()
    {
        TaskEventArgs taskEvent =
            new TaskEventArgs(BuildEventCategory.Custom,
            BuildEventImportance.High, "Important Message",
           "SimpleTask");
        BuildEngine.LogBuildEvent(taskEvent);
        return true;
    }
}
```

Require task parameters to be set

You can mark certain task properties as "required" so that any project file that runs the task must set values for these properties or the build fails. Apply the `[Required]` attribute to the .NET property in your task as follows:

```
[Required]
public string RequiredProperty { get; set; }
```

The `[Required]` attribute is defined by RequiredAttribute in the Microsoft.Build.Framework namespace.

How MSBuild invokes a task

When invoking a task, MSBuild first instantiates the task class, then calls that object's property setters for task parameters that are set in the task element in the project file. If the task element does not specify a parameter, or if the expression specified in the element evaluates to an empty string, the property setter is not called.

For example, in the project

```
<Project>
 <Target Name="InvokeCustomTask">
  <CustomTask Input1=""
              Input2="$(PropertyThatIsNotDefined)"
              Input3="value3" />
 </Target>
</Project>
```

only the setter for `Input3` is called.

A task should not depend on any relative order of parameter-property setter invocation.

Task parameter types

The MSBuild natively handles properties of type `string`, `bool`, `ITaskItem` and `ITaskItem[]`. If a task accepts a parameter of a different type, MSBuild invokes ChangeType to convert from `string` (with all property and item references expanded) to the destination type. If the conversion fails for any input parameter, MSBuild emits an error and does not call the task's `Execute()` method.

Example

Description

This following Visual C# class demonstrates a task deriving from the Task helper class. This task returns `true`, indicating that it succeeded.

Code

```
using System;
using Microsoft.Build.Utilities;

namespace SimpleTask1
{
    public class SimpleTask1: Task
    {
        public override bool Execute()
        {
            // This is where the task would presumably do its work.
            return true;
        }
    }
}
```

Example

Description

This following Visual C# class demonstrates a task implementing the ITask interface. This task returns `true`, indicating that it succeeded.

Code

```
using System;
using Microsoft.Build.Framework;

namespace SimpleTask2
{
    public class SimpleTask2: ITask
    {
        //When implementing the ITask interface, it is necessary to
        //implement a BuildEngine property of type
        //Microsoft.Build.Framework.IBuildEngine. This is done for
        //you if you derive from the Task class.
        public IBuildEngine BuildEngine { get; set; }

        // When implementing the ITask interface, it is necessary to
        // implement a HostObject property of type object.
        // This is done for you if you derive from the Task class.
        public object HostObject { get; set; }

        public bool Execute()
        {
            // This is where the task would presumably do its work.
            return true;
        }
    }
}
```

Example

Description

This Visual C# class demonstrates a task that derives from the Task helper class. It has a required string property, and raises an event that is displayed by all registered loggers.

Code

```
using System;
using Microsoft.Build.Framework;
using Microsoft.Build.Utilities;

namespace SimpleTask3
{
    public class SimpleTask3 : Task
    {
        // The [Required] attribute indicates a required property.
        // If a project file invokes this task without passing a value
        // to this property, the build will fail immediately.
        [Required]
        public string MyProperty { get; set; }

        public override bool Execute()
        {
            // Log a high-importance comment
            Log.LogMessage(MessageImportance.High,
                "The task was passed \"" + MyProperty + "\".");
            return true;
        }
    }
}
```

Example

Description

The following example shows a project file invoking the previous example task, SimpleTask3.

Code

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <UsingTask TaskName="SimpleTask3.SimpleTask3"
        AssemblyFile="SimpleTask3\bin\debug\simpletask3.dll"/>

    <Target Name="MyTarget">
        <SimpleTask3 MyProperty="Hello!"/>
    </Target>
</Project>
```

See also

- Task reference

How to: Ignore errors in tasks

12/4/2019 • 2 minutes to read • Edit Online

Sometimes you want a build to be tolerant of faults in certain tasks. If those non-critical tasks fail, you want the build to continue because it can still produce the required output. For example, if a project uses a `SendMail` task to send an e-mail message after each component is built, you might consider it acceptable for the build to proceed to completion even when the mail servers are unavailable and the status messages cannot be sent. Or, for example, if intermediate files are usually deleted during the build, you might consider it acceptable for the build to proceed to completion even when those files cannot be deleted.

Use the ContinueOnError attribute

The `ContinueOnError` attribute of the `Task` element controls whether a build stops or continues when a task failure occurs. This attribute also controls whether errors are treated as errors or warnings when the build continues.

The `ContinueOnError` attribute can contain one of the following values:

- **WarnAndContinue** or **true**. When a task fails, subsequent tasks in the Target element and the build continue to execute, and all errors from the task are treated as warnings.
- **ErrorAndContinue**. When a task fails, subsequent tasks in the `Target` element and the build continue to execute, and all errors from the task are treated as errors.
- **ErrorAndStop** or **false** (default). When a task fails, the remaining tasks in the `Target` element and the build aren't executed, and the entire `Target` element and the build is considered to have failed.

Versions of the .NET Framework before 4.5 supported only the `true` and `false` values.

The default value of `ContinueOnError` is `ErrorAndStop`. If you set the attribute to `ErrorAndStop`, you make the behavior explicit to anyone who reads the project file.

To ignore an error in a task

Use the `ContinueOnError` attribute of the task. For example:

```
<Delete Files="@(Files)" ContinueOnError="WarnAndContinue"/>
```

Example

The following code example illustrates that the `Build` target still runs and the build is considered a success, even if the `Delete` task fails.

```
<Project DefaultTargets="FakeBuild"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <ItemGroup>
        <Files Include="*.obj"/>
    </ItemGroup>
    <Target Name="Clean">
        <Delete Files="@(Files)" ContinueOnError="WarnAndContinue"/>
    </Target>

    <Target Name="FakeBuild" DependsOnTargets="Clean">
        <Message Text="Building after cleaning..."/>
    </Target>
</Project>
```

See also

- MSBuild
- Task reference
- Tasks

How to: Build a project that has resources

2/21/2019 • 2 minutes to read • Edit Online

If you are building localized versions of a project, all user interface elements must be separated into resource files for the different languages. If the project uses only strings, the resource files can use text files. Alternatively, you can use *.resx* files as the resource files.

Compile resources with MSBuild

The library of common tasks that is provided with MSBuild includes a `GenerateResource` task that you can use to compile resources in either *.resx* or text files. This task includes the `Sources` parameter to specify which resource files to compile and the `OutputResources` parameter to specify names for the output resource files. For more information on the `GenerateResource` task, see GenerateResource task.

To compile resources with MSBuild

1. Identify the project's resource files and pass them to the `GenerateResource` task, either as item lists, or as file names.
2. Specify the `OutputResources` parameter of the `GenerateResource` task, which allows you to set the names for the output resource files.
3. Use the `Output` element of the task to store the value of the `OutputResources` parameter in an item.
4. Use the item created from the `Output` element as an input into another task.

Example

The following code example shows how the `Output` element specifies that the `OutputResources` attribute of the `GenerateResource` task will contain the compiled resource files *alpha.resources* and *beta.resources* and that those two files will be placed inside the `Resources` item list. By identifying those *.resources* files as a collection of items of the same name, you can easily use them as inputs for another task, such as the Csc task.

This task is equivalent to using the **/compile** switch for Resgen.exe:

```
Resgen.exe /compile alpha.resx,alpha.resources /compile beta.txt,beta.resources
```

```
<GenerateResource
    Sources="alpha.resx; beta.txt"
    OutputResources="alpha.resources; beta.resources">
    <Output TaskParameter="OutputResources"
        ItemName="Resources"/>
</GenerateResource>
```

Example

The following example project contains two tasks: the `GenerateResource` task to compile resources and the `Csc` task to compile both the source code files and the compiled resources files. The resource files compiled by the `GenerateResource` task are stored in the `Resources` item and passed to the `Csc` task.

```
<Project DefaultTargets = "Build"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003" >

    <Target Name="Resources">
        <GenerateResource
            Sources="alpha.resx; beta.txt"
            OutputResources="alpha.resources; beta.resources">
            <Output TaskParameter="OutputResources"
                ItemName="Resources"/>
        </GenerateResource>
    </Target>

    <Target Name="Build" DependsOnTargets="Resources">
        <Csc Sources="hello.cs"
                Resources="@(Resources)"
                OutputAssembly="hello.exe"/>
    </Target>
</Project>
```

See also

- MSBuild
- GenerateResource task
- Csc task
- Resgen.exe (Resource File Generator)

MSBuild inline tasks

7/23/2019 • 5 minutes to read • Edit Online

MSBuild tasks are typically created by compiling a class that implements the ITask interface. For more information, see Tasks.

Starting in .NET Framework version 4, you can create tasks inline in the project file. You do not have to create a separate assembly to host the task. This makes it easier to keep track of source code and easier to deploy the task. The source code is integrated into the script.

In MSBuild 15.8, the RoslynCodeTaskFactory was added which can create .NET Standard cross-platform inline tasks. If you need to use inline tasks on .NET Core, you must use the RoslynCodeTaskFactory.

The structure of an inline task

An inline task is contained by a UsingTask element. The inline task and the `UsingTask` element that contains it are typically included in a *.targets* file and imported into other project files as required. Here is a basic inline task. Notice that it does nothing.

```
<Project ToolsVersion="15.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <!-- This simple inline task does nothing. -->
  <UsingTask
    TaskName="DoNothing"
    TaskFactory="CodeTaskFactory"
    AssemblyFile="$(MSBuildToolsPath)\Microsoft.Build.Tasks.Core.dll" >
    <ParameterGroup />
    <Task>
      <Reference Include="" />
      <Using Namespace="" />
      <Code Type="Fragment" Language="cs">
      </Code>
    </Task>
  </UsingTask>
</Project>
```

The `UsingTask` element in the example has three attributes that describe the task and the inline task factory that compiles it.

- The `TaskName` attribute names the task, in this case, `DoNothing`.
- The `TaskFactory` attribute names the class that implements the inline task factory.
- The `AssemblyFile` attribute gives the location of the inline task factory. Alternatively, you can use the `AssemblyName` attribute to specify the fully qualified name of the inline task factory class, which is typically located in the global assembly cache (GAC).

The remaining elements of the `DoNothing` task are empty and are provided to illustrate the order and structure of an inline task. A more robust example is presented later in this topic.

- The `ParameterGroup` element is optional. When specified, it declares the parameters for the task. For more information about input and output parameters, see Input and output parameters later in this topic.
- The `Task` element describes and contains the task source code.
- The `Reference` element specifies references to the .NET assemblies that you are using in your code. This is

equivalent to adding a reference to a project in Visual Studio. The `Include` attribute specifies the path of the referenced assembly.

- The `Using` element lists the namespaces that you want to access. This resembles the `Using` statement in Visual C#. The `Namespace` attribute specifies the namespace to include.

`Reference` and `Using` elements are language-agnostic. Inline tasks can be written in any one of the supported .NET CodeDom languages, for example, Visual Basic or Visual C#.

> **NOTE**
> Elements contained by the `Task` element are specific to the task factory, in this case, the code task factory.

Code element

The last child element to appear within the `Task` element is the `Code` element. The `Code` element contains or locates the code that you want to be compiled into a task. What you put in the `Code` element depends on how you want to write the task.

The `Language` attribute specifies the language in which your code is written. Acceptable values are `cs` for C#, `vb` for Visual Basic.

The `Type` attribute specifies the type of code that is found in the `Code` element.

- If the value of `Type` is `Class`, then the `Code` element contains code for a class that derives from the ITask interface.
- If the value of `Type` is `Method`, then the code defines an override of the `Execute` method of the ITask interface.
- If the value of `Type` is `Fragment`, then the code defines the contents of the `Execute` method, but not the signature or the `return` statement.

The code itself typically appears between a `<![CDATA[` marker and a `]]>` marker. Because the code is in a CDATA section, you do not have to worry about escaping reserved characters, for example, "<" or ">".

Alternatively, you can use the `Source` attribute of the `Code` element to specify the location of a file that contains the code for your task. The code in the source file must be of the type that is specified by the `Type` attribute. If the `Source` attribute is present, the default value of `Type` is `Class`. If `Source` is not present, the default value is `Fragment`.

> **NOTE**
> When defining the task class in the source file, the class name must agree with the `TaskName` attribute of the corresponding UsingTask element.

HelloWorld

Here is a more robust inline task. The HelloWorld task displays "Hello, world!" on the default error logging device, which is typically the system console or the Visual Studio **Output** window. The `Reference` element in the example is included just for illustration.

```
<Project ToolsVersion="15.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <!-- This simple inline task displays "Hello, world!" -->
  <UsingTask
    TaskName="HelloWorld"
    TaskFactory="CodeTaskFactory"
    AssemblyFile="$(MSBuildToolsPath)\Microsoft.Build.Tasks.Core.dll" >
    <ParameterGroup />
    <Task>
      <Reference Include="System.Xml"/>
      <Using Namespace="System"/>
      <Using Namespace="System.IO"/>
      <Code Type="Fragment" Language="cs">
<![CDATA[
// Display "Hello, world!"
Log.LogError("Hello, world!");
]]>
      </Code>
    </Task>
  </UsingTask>
</Project>
```

You could save the HelloWorld task in a file that is named *HelloWorld.targets*, and then invoke it from a project as follows.

```
<Project ToolsVersion="15.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Import Project="HelloWorld.targets" />
  <Target Name="Hello">
    <HelloWorld />
  </Target>
</Project>
```

Input and output parameters

Inline task parameters are child elements of a `ParameterGroup` element. Every parameter takes the name of the element that defines it. The following code defines the parameter `Text`.

```
<ParameterGroup>
  <Text />
</ParameterGroup>
```

Parameters may have one or more of these attributes:

- `Required` is an optional attribute that is `false` by default. If `true`, then the parameter is required and must be given a value before calling the task.
- `ParameterType` is an optional attribute that is `System.String` by default. It may be set to any fully qualified type that is either an item or a value that can be converted to and from a string by using System.Convert.ChangeType. (In other words, any type that can be passed to and from an external task.)
- `Output` is an optional attribute that is `false` by default. If `true`, then the parameter must be given a value before returning from the Execute method.

For example,

```
<ParameterGroup>
  <Expression Required="true" />
  <Files ParameterType="Microsoft.Build.Framework.ITaskItem[]" Required="true" />
  <Tally ParameterType="System.Int32" Output="true" />
</ParameterGroup>
```

defines these three parameters:

- `Expression` is a required input parameter of type System.String.
- `Files` is a required item list input parameter.
- `Tally` is an output parameter of type System.Int32.

If the `Code` element has the `Type` attribute of `Fragment` or `Method`, then properties are automatically created for every parameter. Otherwise, properties must be explicitly declared in the task source code, and must exactly match their parameter definitions.

Example

The following inline task replaces every occurrence of a token in the given file with the given value.

```
<Project xmlns='http://schemas.microsoft.com/developer/msbuild/2003' ToolsVersion="15.0">

  <UsingTask TaskName="TokenReplace" TaskFactory="CodeTaskFactory"
AssemblyFile="$(MSBuildToolsPath)\Microsoft.Build.Tasks.Core.dll">
    <ParameterGroup>
      <Path ParameterType="System.String" Required="true" />
      <Token ParameterType="System.String" Required="true" />
      <Replacement ParameterType="System.String" Required="true" />
    </ParameterGroup>
    <Task>
      <Code Type="Fragment" Language="cs"><![CDATA[
string content = File.ReadAllText(Path);
content = content.Replace(Token, Replacement);
File.WriteAllText(Path, content);

]]></Code>
    </Task>
  </UsingTask>

  <Target Name='Demo' >
    <TokenReplace Path="C:\Project\Target.config" Token="$MyToken$" Replacement="MyValue"/>
  </Target>
</Project>
```

See also

- Tasks
- Walkthrough: Create an inline task

Walkthrough: Create an inline task

10/18/2019 • 7 minutes to read • Edit Online

MSBuild tasks are typically created by compiling a class that implements the ITask interface. Starting with the .NET Framework version 4, you can create tasks inline in the project file. You do not have to create a separate assembly to host the task. For more information, see Inline tasks.

This walkthrough shows how to create and run these inline tasks:

- A task that has no input or output parameters.
- A task that has one input parameter and no output parameters.
- A task that has two input parameters, and one output parameter that returns an MSBuild property.
- A task that has two input parameters, and one output parameter that returns an MSBuild item.

To create and run the tasks, use Visual Studio and the **Visual Studio Command Prompt Window**, as follows:

1. Create an MSBuild project file by using Visual Studio.
2. Modify the project file in Visual Studio to create the inline task.
3. Use the **Command Prompt Window** to build the project and examine the results.

Create and modify an MSBuild project

The Visual Studio project system is based on MSBuild. Therefore, you can create a build project file by using Visual Studio. In this section, you create a Visual C# project file. (You can create a Visual Basic project file instead. In the context of this tutorial, the difference between the two project files is minor.)

To create and modify a project file

1. In Visual Studio, on the **File** menu, click **New** and then click **Project**.
2. In the **New Project** dialog box, select the **Visual C#** project type, and then select the **Windows Forms Application** template. In the **Name** box, type `InlineTasks`. Type a **Location** for the solution, for example, *D:*\. Ensure that **Create directory for solution** is selected, **Add to Source Control** is cleared, and **Solution Name** is **InlineTasks**.
3. Click **OK** to create the project file.
4. In **Solution Explorer**, right-click the **InlineTasks** project node, and then click **Unload Project**.
5. Right-click the project node again, and then click **Edit InlineTasks.csproj**.

   The project file appears in the code editor.

Add a basic Hello task

Now, add to the project file a basic task that displays the message "Hello, world!" Also add a default TestBuild target to invoke the task.

To add a basic Hello task

1. In the root `Project` node, change the `DefaultTargets` attribute to `TestBuild`.The resulting `Project` node should resemble this example:

```
<Project ToolsVersion="4.0" DefaultTargets="TestBuild"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
```

2. Add the following inline task and target to the project file just before the `</Project>` tag.

```
<UsingTask TaskName="Hello" TaskFactory="CodeTaskFactory"
AssemblyFile="$(MSBuildToolsPath)\Microsoft.Build.Tasks.v4.0.dll" >
  <ParameterGroup />
  <Task>
    <Code Type="Fragment" Language="cs">
      Log.LogMessage("Hello, world!", MessageImportance.High);
    </Code>
  </Task>
</UsingTask>
<Target Name="TestBuild">
  <Hello />
</Target>
```

3. Save the project file.

   This code creates an inline task that is named Hello and has no parameters, references, or `Using` directives. The Hello task contains just one line of code, which displays a hello message on the default logging device, typically the console window.

Run the Hello task

Run MSBuild by using the **Command Prompt Window** to construct the Hello task and to process the TestBuild target that invokes it.

To run the Hello task

1. Click **Start**, click **All Programs**, and then locate the **Visual Studio Tools** folder and click **Visual Studio Command Prompt**.
2. In the **Command Prompt Window**, locate the folder that contains the project file, in this case, *D:\InlineTasks\InlineTasks*.
3. Type **msbuild** without command switches, and then press **Enter**. By default, this builds the *InlineTasks.csproj* file and processes the default target TestBuild, which invokes the Hello task.
4. Examine the output in the **Command Prompt Window**. You should see this line:

   `Hello, world!`

   > **NOTE**
   > If you do not see the hello message, try saving the project file again and then run the Hello task.

   By alternating between the code editor and the **Command Prompt Window**, you can change the project file and quickly see the results.

Define the Echo task

Create an inline task that accepts a string parameter and displays the string on the default logging device.

To define the Echo task

1. In the code editor, replace the Hello task and TestBuild target by using the following code.

```
<UsingTask TaskName="Echo" TaskFactory="CodeTaskFactory"
AssemblyFile="$(MSBuildToolsPath)\Microsoft.Build.Tasks.v4.0.dll" >
  <ParameterGroup>
    <Text Required="true" />
  </ParameterGroup>
  <Task>
    <Code Type="Fragment" Language="cs">
      Log.LogMessage(Text, MessageImportance.High);
    </Code>
  </Task>
</UsingTask>
<Target Name="TestBuild">
  <Echo Text="Greetings!" />
</Target>
```

2. In the **Command Prompt Window**, type **msbuild** without command switches, and then press **Enter**. By default, this processes the default target TestBuild, which invokes the Echo task.

3. Examine the output in the **Command Prompt Window**. You should see this line:

   ```
   Greetings!
   ```

   This code defines an inline task that is named Echo and has just one required input parameter Text. By default, parameters are of type System.String. The value of the Text parameter is set when the TestBuild target invokes the Echo task.

Define the Adder task

Create an inline task that adds two integer parameters and emits their sum as an MSBuild property.

To define the Adder task

1. In the code editor, replace the Echo task and TestBuild target by using the following code.

```
<UsingTask TaskName="Adder" TaskFactory="CodeTaskFactory"
AssemblyFile="$(MSBuildToolsPath)\Microsoft.Build.Tasks.v4.0.dll" >
  <ParameterGroup>
    <A ParameterType="System.Int32" Required="true" />
    <B ParameterType="System.Int32" Required="true" />
    <C ParameterType="System.Int32" Output="true" />
  </ParameterGroup>
  <Task>
    <Code Type="Fragment" Language="cs">
      C = A + B;
    </Code>
  </Task>
</UsingTask>
<Target Name="TestBuild">
  <Adder A="4" B="5">
    <Output PropertyName="Sum" TaskParameter="C" />
  </Adder>
  <Message Text="The sum is $(Sum)" Importance="High" />
</Target>
```

2. In the **Command Prompt Window**, type **msbuild** without command switches, and then press **Enter**. By default, this processes the default target TestBuild, which invokes the Echo task.

3. Examine the output in the **Command Prompt Window**. You should see this line:

   ```
   The sum is 9
   ```

   This code defines an inline task that is named Adder and has two required integer input parameters, A and

B, and one integer output parameter, C. The Adder task adds the two input parameters and returns the sum in the output parameter. The sum is emitted as the MSBuild property `Sum`. The values of the input parameters are set when the TestBuild target invokes the Adder task.

Define the RegX task

Create an inline task that accepts an item group and a regular expression, and returns a list of all items that have file content that matches the expression.

To define the RegX task

1. In the code editor, replace the Adder task and TestBuild target by using the following code.

```
<UsingTask TaskName="RegX" TaskFactory="CodeTaskFactory"
AssemblyFile="$(MSBuildToolsPath)\Microsoft.Build.Tasks.v4.0.dll" >
  <ParameterGroup>
    <Expression Required="true" />
    <Files ParameterType="Microsoft.Build.Framework.ITaskItem[]" Required="true" />
    <Result ParameterType="Microsoft.Build.Framework.ITaskItem[]" Output="true" />
  </ParameterGroup>
  <Task>
    <Using Namespace="System.Text.RegularExpressions"/>
    <Code Type="Fragment" Language="cs">
<![CDATA[
      if (Files.Length > 0)
      {
        Result = new TaskItem[Files.Length];
        for (int i = 0; i < Files.Length; i++)
        {
          ITaskItem item = Files[i];
          string path = item.GetMetadata("FullPath");
          using(StreamReader rdr = File.OpenText(path))
          {
            if (Regex.Match(rdr.ReadToEnd(), Expression).Success)
            {
              Result[i] = new TaskItem(item.ItemSpec);
            }
          }
        }
      }
]]>
    </Code>
  </Task>
</UsingTask>
<Target Name="TestBuild">
  <RegX Expression="public|protected" Files="@(Compile)">
    <Output ItemName="MatchedFiles" TaskParameter="Result" />
  </RegX>
  <Message Text="Input files: @(Compile)" Importance="High" />
  <Message Text="Matched files: @(MatchedFiles)" Importance="High" />
</Target>
```

2. In the **Command Prompt Window**, type **msbuild** without command switches, and then press **Enter**. By default, this processes the default target TestBuild, which invokes the RegX task.

3. Examine the output in the **Command Prompt Window**. You should see these lines:

```
Input files:
Form1.cs;Form1.Designer.cs;Program.cs;Properties\AssemblyInfo.cs;Properties\Resources.Designer.cs;Proper
ties\Settings.Designer.cs
```

```
Matched files: Form1.cs;Form1.Designer.cs;Properties\Settings.Designer.cs
```

This code defines an inline task that is named RegX and has these three parameters:

- `Expression` is a required string input parameter that has a value that is the regular expression to be matched. In this example, the expression matches the words "public" or "protected".
- `Files` is a required item list input parameter that has a value that is a list of files to be searched for the match. In this example, `Files` is set to the `Compile` item, which lists the project source files.
- `Result` is an output parameter that has a value that is the list of files that have contents that match the regular expression.

  The value of the input parameters are set when the TestBuild target invokes the RegX task. The RegX task reads every file and returns the list of files that match the regular expression. This list is returned as the `Result` output parameter, which is emitted as the MSBuild item `MatchedFiles`.

Handle reserved characters

The MSBuild parser processes inline tasks as XML. Characters that have reserved meaning in XML, for example, "<" and ">", are detected and handled as if they were XML, and not .NET source code. To include the reserved characters in code expressions such as `Files.Length > 0`, write the `Code` element so that its contents are contained in a CDATA expression, as follows:

```
<Code Type="Fragment" Language="cs">
 <![CDATA[

 // Your code goes here.

 ]]>
</Code>
```

See also

- Inline tasks
- Tasks
- Targets

MSBuild inline tasks with RoslynCodeTaskFactory

4/23/2019 • 6 minutes to read • Edit Online

Similar to the CodeTaskFactory, RoslynCodeTaskFactory uses the cross-platform Roslyn compilers to generate in-memory task assemblies for use as inline tasks. RoslynCodeTaskFactory tasks target .NET Standard and can work on .NET Framework and .NET Core runtimes as well as other platforms such as Linux and Mac OS.

> **NOTE**
> The RoslynCodeTaskFactory is available in MSBuild 15.8 and above only.

The structure of an inline task with RoslynCodeTaskFactory

RoslynCodeTaskFactory inline tasks are declared in an identical way as CodeTaskFactory, the only difference being that they target .NET Standard. The inline task and the `UsingTask` element that contains it are typically included in a *.targets* file and imported into other project files as required. Here is a basic inline task. Notice that it does nothing.

```
<Project ToolsVersion="15.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <!-- This simple inline task does nothing. -->
  <UsingTask
    TaskName="DoNothing"
    TaskFactory="RoslynCodeTaskFactory"
    AssemblyFile="$(MSBuildToolsPath)\Microsoft.Build.Tasks.Core.dll" >
    <ParameterGroup />
    <Task>
      <Reference Include="" />
      <Using Namespace="" />
      <Code Type="Fragment" Language="cs">
      </Code>
    </Task>
  </UsingTask>
</Project>
```

The `UsingTask` element in the example has three attributes that describe the task and the inline task factory that compiles it.

- The `TaskName` attribute names the task, in this case, `DoNothing`.
- The `TaskFactory` attribute names the class that implements the inline task factory.
- The `AssemblyFile` attribute gives the location of the inline task factory. Alternatively, you can use the `AssemblyName` attribute to specify the fully qualified name of the inline task factory class, which is typically located in the global assembly cache (GAC).

The remaining elements of the `DoNothing` task are empty and are provided to illustrate the order and structure of an inline task. A more robust example is presented later in this topic.

- The `ParameterGroup` element is optional. When specified, it declares the parameters for the task. For more information about input and output parameters, see Input and Output Parameters later in this topic.
- The `Task` element describes and contains the task source code.
- The `Reference` element specifies references to the .NET assemblies that you are using in your code. This is

equivalent to adding a reference to a project in Visual Studio. The `Include` attribute specifies the path of the referenced assembly.

- The `Using` element lists the namespaces that you want to access. This resembles the `Using` statement in Visual C#. The `Namespace` attribute specifies the namespace to include.

`Reference` and `Using` elements are language-agnostic. Inline tasks can be written in any one of the supported .NET CodeDom languages, for example, Visual Basic or Visual C#.

> **NOTE**
> Elements contained by the `Task` element are specific to the task factory, in this case, the code task factory.

Code element

The last child element to appear within the `Task` element is the `Code` element. The `Code` element contains or locates the code that you want to be compiled into a task. What you put in the `Code` element depends on how you want to write the task.

The `Language` attribute specifies the language in which your code is written. Acceptable values are `cs` for C#, `vb` for Visual Basic.

The `Type` attribute specifies the type of code that is found in the `Code` element.

- If the value of `Type` is `Class`, then the `Code` element contains code for a class that derives from the ITask interface.
- If the value of `Type` is `Method`, then the code defines an override of the `Execute` method of the ITask interface.
- If the value of `Type` is `Fragment`, then the code defines the contents of the `Execute` method, but not the signature or the `return` statement.

The code itself typically appears between a `<![CDATA[` marker and a `]]>` marker. Because the code is in a CDATA section, you do not have to worry about escaping reserved characters, for example, "<" or ">".

Alternatively, you can use the `Source` attribute of the `Code` element to specify the location of a file that contains the code for your task. The code in the source file must be of the type that is specified by the `Type` attribute. If the `Source` attribute is present, the default value of `Type` is `Class`. If `Source` is not present, the default value is `Fragment`.

> **NOTE**
> When defining the task class in the source file, the class name must agree with the `TaskName` attribute of the corresponding UsingTask element.

Hello World

Here is a more robust inline task with RoslynCodeTaskFactory. The HelloWorld task displays "Hello, world!" on the default error logging device, which is typically the system console or the Visual Studio **Output** window. The `Reference` element in the example is included just for illustration.

```
<Project ToolsVersion="15.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <!-- This simple inline task displays "Hello, world!" -->
  <UsingTask
    TaskName="HelloWorld"
    TaskFactory="RoslynCodeTaskFactory"
    AssemblyFile="$(MSBuildToolsPath)\Microsoft.Build.Tasks.Core.dll" >
    <ParameterGroup />
    <Task>
      <Reference Include="System.Xml"/>
      <Using Namespace="System"/>
      <Using Namespace="System.IO"/>
      <Code Type="Fragment" Language="cs">
<![CDATA[
// Display "Hello, world!"
Log.LogError("Hello, world!");
]]>
      </Code>
    </Task>
  </UsingTask>
</Project>
```

You could save the HelloWorld task in a file that is named *HelloWorld.targets*, and then invoke it from a project as follows.

```
<Project ToolsVersion="15.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <Import Project="HelloWorld.targets" />
  <Target Name="Hello">
    <HelloWorld />
  </Target>
</Project>
```

Input and output parameters

Inline task parameters are child elements of a `ParameterGroup` element. Every parameter takes the name of the element that defines it. The following code defines the parameter `Text`.

```
<ParameterGroup>
    <Text />
</ParameterGroup>
```

Parameters may have one or more of these attributes:

- `Required` is an optional attribute that is `false` by default. If `true`, then the parameter is required and must be given a value before calling the task.
- `ParameterType` is an optional attribute that is `System.String` by default. It may be set to any fully qualified type that is either an item or a value that can be converted to and from a string by using System.Convert.ChangeType. (In other words, any type that can be passed to and from an external task.)
- `Output` is an optional attribute that is `false` by default. If `true`, then the parameter must be given a value before returning from the Execute method.

For example,

```
<ParameterGroup>
    <Expression Required="true" />
    <Files ParameterType="Microsoft.Build.Framework.ITaskItem[]" Required="true" />
    <Tally ParameterType="System.Int32" Output="true" />
</ParameterGroup>
```

defines these three parameters:

- `Expression` is a required input parameter of type System.String.
- `Files` is a required item list input parameter.
- `Tally` is an output parameter of type System.Int32.

If the `Code` element has the `Type` attribute of `Fragment` or `Method`, then properties are automatically created for every parameter. Otherwise, properties must be explicitly declared in the task source code, and must exactly match their parameter definitions.

Example

The following inline task logs some messages and returns a string.

```
<Project xmlns='http://schemas.microsoft.com/developer/msbuild/2003' ToolsVersion="15.0">

    <UsingTask TaskName="MySample"
               TaskFactory="RoslynCodeTaskFactory"
               AssemblyFile="$(MSBuildBinPath)\Microsoft.Build.Tasks.Core.dll">
        <ParameterGroup>
            <Parameter1 ParameterType="System.String" Required="true" />
            <Parameter2 ParameterType="System.String" />
            <Parameter3 ParameterType="System.String" Output="true" />
        </ParameterGroup>
        <Task>
            <Using Namespace="System" />
            <Code Type="Fragment" Language="C#">
              <![CDATA[
              Log.LogMessage(MessageImportance.High, "Hello from an inline task created by Roslyn!");
              Log.LogMessageFromText($"Parameter1: '{Parameter1}'", MessageImportance.High);
              Log.LogMessageFromText($"Parameter2: '{Parameter2}'", MessageImportance.High);
              Parameter3 = "A value from the Roslyn CodeTaskFactory";
            ]]>
            </Code>
        </Task>
    </UsingTask>

    <Target Name="Demo">
      <MySample Parameter1="A value for parameter 1" Parameter2="A value for parameter 2">
          <Output TaskParameter="Parameter3" PropertyName="NewProperty" />
      </MySample>

      <Message Text="NewProperty: '$(NewProperty)'" />
    </Target>
</Project>
```

These inline tasks can combine paths and get the file name.

```
<Project xmlns='http://schemas.microsoft.com/developer/msbuild/2003' ToolsVersion="15.0">

    <UsingTask TaskName="PathCombine"
               TaskFactory="RoslynCodeTaskFactory"
               AssemblyFile="$(MSBuildBinPath)\Microsoft.Build.Tasks.Core.dll">
        <ParameterGroup>
            <Paths ParameterType="System.String[]" Required="true" />
            <Combined ParameterType="System.String" Output="true" />
        </ParameterGroup>
        <Task>
            <Using Namespace="System" />
            <Code Type="Fragment" Language="C#">
            <![CDATA[
            Combined = Path.Combine(Paths);
            ]]>
            </Code>
        </Task>
    </UsingTask>

    <UsingTask TaskName="PathGetFileName"
             TaskFactory="RoslynCodeTaskFactory"
             AssemblyFile="$(MSBuildBinPath)\Microsoft.Build.Tasks.Core.dll">
        <ParameterGroup>
            <Path ParameterType="System.String" Required="true" />
            <FileName ParameterType="System.String" Output="true" />
        </ParameterGroup>
        <Task>
            <Using Namespace="System" />
            <Code Type="Fragment" Language="C#">
            <![CDATA[
            FileName = System.IO.Path.GetFileName(Path);
            ]]>
            </Code>
        </Task>
    </UsingTask>

    <Target Name="Demo">
        <PathCombine Paths="$(Temp);MyFolder;$([System.Guid]::NewGuid()).txt">
            <Output TaskParameter="Combined" PropertyName="MyCombinedPaths" />
        </PathCombine>

        <Message Text="Combined Paths: '$(MyCombinedPaths)'" />

        <PathGetFileName Path="$(MyCombinedPaths)">
            <Output TaskParameter="FileName" PropertyName="MyFileName" />
        </PathGetFileName>

        <Message Text="File name: '$(MyFileName)'" />
    </Target>
</Project>
```

See also

- Tasks
- Walkthrough: Create an inline task

Compare properties and items

7/11/2019 • 5 minutes to read • Edit Online

MSBuild properties and items are both used to pass information to tasks, evaluate conditions, and store values that can be referenced throughout the project file.

- Properties are name-value pairs. For more information, see MSBuild properties.
- Items are objects that typically represent files. Item objects can have associated metadata collections. Metadata are name-value pairs. For more information, see Items.

Scalars and vectors

Because MSBuild properties are name-value pairs that have just one string value, they are often described as *scalar*. Because MSBuild item types are lists of items, they are often described as *vector*. However, in practice, properties can represent multiple values, and item types can have zero or one items.

Target dependency injection

To see how properties can represent multiple values, consider a common usage pattern for adding a target to a list of targets to be built. This list is typically represented by a property value, with the target names separated by semicolons.

```
<PropertyGroup>
    <BuildDependsOn>
        BeforeBuild;
        CoreBuild;
        AfterBuild
    </BuildDependsOn>
</PropertyGroup>
```

The `BuildDependsOn` property is typically used as the argument of a target `DependsOnTargets` attribute, effectively converting it to an item list. This property can be overridden to add a target or to change the target execution order. For example,

```
<PropertyGroup>
    <BuildDependsOn>
        $(BuildDependsOn);
        CustomBuild;
    </BuildDependsOn>
</PropertyGroup>
```

adds the CustomBuild target to the target list, giving `BuildDependsOn` the value `BeforeBuild;CoreBuild;AfterBuild;CustomBuild`.

Starting with MSBuild 4.0, target dependency injection is deprecated. Use the `AfterTargets` and `BeforeTargets` attributes instead. For more information, see Target build order.

Conversions between strings and item lists

MSBuild performs conversions to and from item types and string values as needed. To see how an item list can become a string value, consider what happens when an item type is used as the value of an MSBuild property:

```
<ItemGroup>
    <OutputDir Include="KeyFiles\;Certificates\" />
</ItemGroup>
<PropertyGroup>
    <OutputDirList>@(OutputDir)</OutputDirList>
</PropertyGroup>
```

The item type OutputDir has an `Include` attribute with the value "KeyFiles\;Certificates\". MSBuild parses this string into two items: KeyFiles\ and Certificates\. When the item type OutputDir is used as the value of the OutputDirList property, MSBuild converts or "flattens" the item type into to the semicolon-separated string "KeyFiles\;Certificates\".

Properties and items in tasks

Properties and items are used as inputs and outputs to MSBuild tasks. For more information, see Tasks.

Properties are passed to tasks as attributes. Within the task, an MSBuild property is represented by a property type whose value can be converted to and from a string. The supported property types include `bool`, `char`, `DateTime`, `Decimal`, `Double`, `int`, `string`, and any type that ChangeType can handle.

Items are passed to tasks as ITaskItem objects. Within the task, ItemSpec represents the value of the item and GetMetadata retrieves its metadata.

The item list of an item type can be passed as an array of `ITaskItem` objects. Beginning with the .NET Framework 3.5, items can be removed from an item list in a target by using the `Remove` attribute. Because items can be removed from an item list, it is possible for an item type to have zero items. If an item list is passed to a task, the code in the task should check for this possibility.

Property and item evaluation order

During the evaluation phase of a build, imported files are incorporated into the build in the order in which they appear. Properties and items are defined in three passes in the following order:

- Properties are defined and modified in the order in which they appear.
- Item definitions are defined and modified in the order in which they appear.
- Items are defined and modified in the order in which they appear.

During the execution phase of a build, properties and items that are defined within targets are evaluated together in a single phase in the order in which they appear.

However, this is not the full story. When a property, item definition, or item is defined, its value is evaluated. The expression evaluator expands the string that specifies the value. The string expansion is dependent on the build phase. Here is a more detailed property and item evaluation order:

- During the evaluation phase of a build:
    - Properties are defined and modified in the order in which they appear. Property functions are executed. Property values in the form $(PropertyName) are expanded within expressions. The property value is set to the expanded expression.
    - Item definitions are defined and modified in the order in which they appear. Property functions have already been expanded within expressions. Metadata values are set to the expanded expressions.
    - Item types are defined and modified in the order in which they appear. Item values in the form @(ItemType) are expanded. Item transformations are also expanded. Property functions and values have already been expanded within expressions. The item list and metadata values are set to the

expanded expressions.

- During the execution phase of a build:
  - Properties and items that are defined within targets are evaluated together in the order in which they appear. Property functions are executed and property values are expanded within expressions. Item values and item transformations are also expanded. The property values, item type values, and metadata values are set to the expanded expressions.

Subtle effects of the evaluation order

In the evaluation phase of a build, property evaluation precedes item evaluation. Nevertheless, properties can have values that appear to depend on item values. Consider the following script.

```
<ItemGroup>
    <KeyFile Include="KeyFile.cs">
        <Version>1.0.0.3</Version>
    </KeyFile>
</ItemGroup>
<PropertyGroup>
    <KeyFileVersion>@(KeyFile->'%(Version)')</KeyFileVersion>
</PropertyGroup>
<Target Name="AfterBuild">
    <Message Text="KeyFileVersion: $(KeyFileVersion)" />
</Target>
```

Executing the Message task displays this message:

```
KeyFileVersion: 1.0.0.3
```

This is because the value of `KeyFileVersion` is actually the string "@(KeyFile->'%(Version)')". Item and item transformations were not expanded when the property was first defined, so the `KeyFileVersion` property was assigned the value of the unexpanded string.

During the execution phase of the build, when it processes the Message task, MSBuild expands the string "@(KeyFile->'%(Version)')" to yield "1.0.0.3".

Notice that the same message would appear even if the property and item groups were reversed in order.

As a second example, consider what can happen when property and item groups are located within targets:

```
<Target Name="AfterBuild">
    <PropertyGroup>
        <KeyFileVersion>@(KeyFile->'%(Version)')</KeyFileVersion>
    </PropertyGroup>
    <ItemGroup>
        <KeyFile Include="KeyFile.cs">
            <Version>1.0.0.3</Version>
        </KeyFile>
    </ItemGroup>
    <Message Text="KeyFileVersion: $(KeyFileVersion)" />
</Target>
```

The Message task displays this message:

```
KeyFileVersion:
```

This is because during the execution phase of the build, property and item groups defined within targets are

evaluated top to bottom at the same time. When `KeyFileVersion` is defined, `KeyFile` is unknown. Therefore, the item transformation expands to an empty string.

In this case, reversing the order of the property and item groups restores the original message:

```
<Target Name="AfterBuild">
    <ItemGroup>
        <KeyFile Include="KeyFile.cs">
            <Version>1.0.0.3</Version>
        </KeyFile>
    </ItemGroup>
    <PropertyGroup>
        <KeyFileVersion>@(KeyFile->'%(Version)')</KeyFileVersion>
    </PropertyGroup>
    <Message Text="KeyFileVersion: $(KeyFileVersion)" />
</Target>
```

The value of `KeyFileVersion` is set to "1.0.0.3" and not to "@(KeyFile->'%(Version)')". The Message task displays this message:

```
KeyFileVersion: 1.0.0.3
```

See also

- Advanced concepts

MSBuild special characters

6/18/2019 • 2 minutes to read • Edit Online

MSBuild reserves some characters for special use in specific contexts. You only have to escape such characters if you want to use them literally in the context in which they are reserved. For example, an asterisk has special meaning only in the `Include` and `Exclude` attributes of an item definition, and in calls to `CreateItem`. If you want an asterisk to appear as an asterisk in one of those contexts, you must escape it. In every other context, just type the asterisk where you want it to appear.

To escape a special character, use the syntax %<xx>, where <xx> represents the ASCII hexadecimal value of the character. For more information, see How to: Escape special characters in MSBuild.

Special characters

The following table lists MSBuild special characters:

CHARACTER	ASCII	RESERVED USAGE
%	%25	Referencing metadata
$	%24	Referencing properties
@	%40	Referencing item lists
'	%27	Conditions and other expressions
;	%3B	List separator
?	%3F	Wildcard character for file names in `Include` and `Exclude` attributes
*	%2A	Wildcard character for use in file names in `Include` and `Exclude` attributes

See also

- Advanced concepts
- Items

How to: Escape special characters in MSBuild

2/21/2019 • 2 minutes to read • Edit Online

Certain characters have special meaning in MSBuild project files. Examples of the characters include semicolons (`;`) and asterisks (`*`). For a complete list of these special characters, see MSBuild special characters.

In order to use these special characters as literals in a project file, they must be specified by using the syntax `%<xx>`, where `<xx>` represents the ASCII hexadecimal value of the character.

MSBuild special characters

One example of where special characters are used is in the `Include` attribute of item lists. For example, the following item list declares two items: *MyFile.cs* and *MyClass.cs*.

```
<Compile Include="MyFile.cs;MyClass.cs"/>
```

If you want to declare an item that contains a semicolon in the name, you must use the `%<xx>` syntax to escape the semicolon and prevent MSBuild from declaring two separate items. For example, the following item escapes the semicolon and declares one item named `MyFile.cs;MyClass.cs`.

```
<Compile Include="MyFile.cs%3BMyClass.cs"/>
```

You can also use a property function to escape strings. For example, this is equivalent to the example above.

```
<Compile Include="$([MSBuild]::Escape('MyFile.cs;MyClass.cs'))" />
```

To use an MSBuild special character as a literal character

Use the notation `%<xx>` in place of the special character, where `<xx>` represents the hexadecimal value of the ASCII character. For example, to use an asterisk (`*`) as a literal character, use the value `%2A`.

See also

- MSBuild concepts
- MSBuild
- Items

How to: Use reserved XML characters in project files

4/18/2019 • 2 minutes to read • Edit Online

When you author project files, you might need to use reserved XML characters, for example, in property values or in task parameter values. However, some reserved characters must be replaced by a named entity so that the project file can be parsed.

Use reserved characters

The following table describes the reserved XML characters that must be replaced by the corresponding named entity so that the project file can be parsed.

RESERVED CHARACTER	NAMED ENTITY
<	&lt;
>	&gt;
&	&amp;
"	&quot;
'	&apos;

To use double quotes in a project file

- Replace the double quotes with the corresponding named entity, ". For example, to place double quotes around the `EXEFile` item list, type:

```
<Message Text="The output file is &quot;@(EXEFile)&quot;."/>
```

Example

In the following code example, double quotes are used to highlight the file name in the message that is output by the project file.

```
<Project DefaultTargets="Compile"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003" >
    <!-- Set the application name as a property -->
    <PropertyGroup>
        <appname>"HelloWorldCS"</appname>
    </PropertyGroup>
    <!-- Specify the inputs -->
    <ItemGroup>
        <CSFile Include = "consolehwcs1.cs" />
    </ItemGroup>
    <Target Name = "Compile">
        <!-- Run the Visual C# compilation using input
        files of type CSFile -->
        <Csc Sources = "@(CSFile)">
            <!-- Set the OutputAssembly attribute of the CSC task
            to the name of the executable file that is created -->
            <Output
                TaskParameter = "OutputAssembly"
                ItemName = "EXEFile"/>
        </Csc>
        <!-- Log the file name of the output file -->
        <Message Text="The output file is &quot;@(EXEFile)&quot;."/>
    </Target>
</Project>
```

See also

- MSBuild reference
- MSBuild

MSBuild advanced concepts

2/21/2019 • 2 minutes to read • Edit Online

The documents in this section describe how to use advanced techniques to improve builds that you run by using MSBuild.

Related topics

TITLE	DESCRIPTION
Batching	Describes how to batch build targets and tasks based on item metadata.
Transforms	Explains how to use transforms to enable dependency analysis.
Visual Studio integration	Discusses how to use MSBuild project files when you compile code from the Visual Studio IDE.
Build multiple projects in parallel	Describes how to build multiple projects faster on computers that have multiple processors or multicore processors.
Multitargeting	Describes how to compile an application to run on any one of several versions of the .NET Framework.
Best practices	Recommends best practices for writing MSBuild scripts.

See also

- MSBuild concepts
- Logging in MSBuild

MSBuild batching

2/21/2019 • 2 minutes to read • Edit Online

MSBuild has the ability to divide item lists into different categories, or batches, based on item metadata, and run a target or task one time with each batch.

Task batching

Task batching allows you to simplify your project files by providing a way to divide item lists into different batches and pass each of those batches into a task separately. This means that a project file only needs to have the task and its attributes declared once, even though it can be run several times.

You specify that you want MSBuild to perform batching with a task by using the %(<ItemMetaDataName>) notation in one of the task attributes. The following example splits the `Example` item list into batches based on the `Color` item metadata value, and passes each of the batches to the `MyTask` task separately.

> **NOTE**
> If you do not reference the item list elsewhere in the task attributes, or the metadata name may be ambiguous, you can use the %(<ItemCollection.ItemMetaDataName>) notation to fully qualify the item metadata value to use for batching.

```
<Project
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <ItemGroup>
        <Example Include="Item1">
            <Color>Blue</Color>
        </Example>
        <Example Include="Item2">
            <Color>Red</Color>
        </Example>
    </ItemGroup>

    <Target Name="RunMyTask">
        <MyTask
            Sources = "@(Example)"
            Output = "%(Color)\MyFile.txt"/>
    </Target>

</Project>
```

For more specific batching examples, see Item metadata in task batching.

Target batching

MSBuild checks if the inputs and outputs of a target are up-to-date before it runs the target. If both inputs and outputs are up-to-date, the target is skipped. If a task inside of a target uses batching, MSBuild needs to determine if the inputs and outputs for each batch of items is up-to-date. Otherwise, the target is executed every time it is hit.

The following example shows a `Target` element that contains an `Outputs` attribute with the %(<ItemMetaDataName>) notation. MSBuild will divide the `Example` item list into batches based on the `Color` item metadata, and analyze the timestamps of the output files for each batch. If the outputs from a batch are not

up-to-date, the target is run. Otherwise, the target is skipped.

```
<Project
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <ItemGroup>
        <Example Include="Item1">
            <Color>Blue</Color>
        </Example>
        <Example Include="Item2">
            <Color>Red</Color>
        </Example>
    </ItemGroup>

    <Target Name="RunMyTask"
        Inputs="@(Example)"
        Outputs="%(Color)\MyFile.txt">
        <MyTask
            Sources = "@(Example)"
            Output = "%(Color)\MyFile.txt"/>
    </Target>

</Project>
```

For another example of target batching, see Item metadata in target batching.

Property functions using metadata

Batching can be controlled by property functions that include metadata. For example,

`$([System.IO.Path]::Combine($(RootPath),%(Compile.Identity)))`

uses Combine to combine a root folder path with a Compile item path.

Property functions may not appear within metadata values. For example,

`%(Compile.FullPath.Substring(0,3))`

is not allowed.

For more information about property functions, see Property functions.

See also

- ItemMetadata element (MSBuild)
- MSBuild concepts
- MSBuild reference
- Advanced concepts

Item metadata in task batching

12/3/2019 • 4 minutes to read • Edit Online

MSBuild has the ability to divide item lists into different categories, or batches, based on item metadata, and run a task one time with each batch. It can be confusing to understand exactly what items are being passed with which batch. This topic covers the following common scenarios that involve batching.

- Dividing an item list into batches
- Dividing several item lists into batches
- Batching one item at a time
- Filtering item lists

For more information on batching with MSBuild, see Batching.

Divide an item list into batches

Batching allows you to divide an item list into different batches based on item metadata, and pass each of the batches into a task separately. This is useful for building satellite assemblies.

The following example shows how to divide an item list into batches based on item metadata. The `ExampColl` item list is divided into three batches based on the `Number` item metadata. The presence of `%(ExampColl.Number)` in the `Text` attribute notifies MSBuild that batching should be performed. The `ExampColl` item list is divided into three batches based on the `Number` metadata, and each batch is passed separately into the task.

```
<Project
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <ItemGroup>
        <ExampColl Include="Item1">
            <Number>1</Number>
        </ExampColl>
        <ExampColl Include="Item2">
            <Number>2</Number>
        </ExampColl>
        <ExampColl Include="Item3">
            <Number>3</Number>
        </ExampColl>
        <ExampColl Include="Item4">
            <Number>1</Number>
        </ExampColl>
        <ExampColl Include="Item5">
            <Number>2</Number>
        </ExampColl>
        <ExampColl Include="Item6">
            <Number>3</Number>
        </ExampColl>
    </ItemGroup>

    <Target Name="ShowMessage">
        <Message
            Text = "Number: %(ExampColl.Number) -- Items in ExampColl: @(ExampColl)"/>
    </Target>

</Project>
```

The Message task displays the following information:

```
Number: 1 -- Items in ExampColl: Item1;Item4
```

```
Number: 2 -- Items in ExampColl: Item2;Item5
```

```
Number: 3 -- Items in ExampColl: Item3;Item6
```

Divide several item lists into batches

MSBuild can divide multiple item lists into batches based on the same metadata. This makes it easy to divide different item lists into batches to build multiple assemblies. For example, you could have an item list of *.cs* files divided into an application batch and an assembly batch, and an item list of resource files divided into an application batch and an assembly batch. You could then use batching to pass these item lists into one task and build both the application and the assembly.

> **NOTE**
> If an item list being passed into a task contains no items with the referenced metadata, every item in that item list is passed into every batch.

The following example shows how to divide multiple item list into batches based on item metadata. The `ExampColl` and `ExampColl2` item lists are each divided into three batches based on the `Number` item metadata. The presence of `%(Number)` in the `Text` attribute notifies MSBuild that batching should be performed. The `ExampColl` and `ExampColl2` item lists are divided into three batches based on the `Number` metadata, and each batch is passed separately into the task.

```
<Project
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <ItemGroup>

        <ExampColl Include="Item1">
            <Number>1</Number>
        </ExampColl>
        <ExampColl Include="Item2">
            <Number>2</Number>
        </ExampColl>
        <ExampColl Include="Item3">
            <Number>3</Number>
        </ExampColl>

        <ExampColl2 Include="Item4">
            <Number>1</Number>
        </ExampColl2>
        <ExampColl2 Include="Item5">
            <Number>2</Number>
        </ExampColl2>
        <ExampColl2 Include="Item6">
            <Number>3</Number>
        </ExampColl2>

    </ItemGroup>

    <Target Name="ShowMessage">
        <Message
            Text = "Number: %(Number) -- Items in ExampColl: @(ExampColl) ExampColl2: @(ExampColl2)"/>
    </Target>

</Project>
```

The Message task displays the following information:

```
Number: 1 -- Items in ExampColl: Item1 ExampColl2: Item4
```

```
Number: 2 -- Items in ExampColl: Item2 ExampColl2: Item5
```

```
Number: 3 -- Items in ExampColl: Item3 ExampColl2: Item6
```

Batch one item at a time

Batching can also be performed on well-known item metadata that is assigned to every item upon creation. This guarantees that every item in a collection will have some metadata to use for batching. The `Identity` metadata value is unique for every item, and is useful for dividing every item in an item list into a separate batch. For a complete list of well-known item metadata, see Well-known item metadata.

The following example shows how to batch each item in an item list one at a time. Because the `Identity` metadata value of every item is unique, the `ExampColl` item list is divided into six batches, each batch containing one item of the item list. The presence of `%(Identity)` in the `Text` attribute notifies MSBuild that batching should be performed.

```
<Project
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <ItemGroup>

        <ExampColl Include="Item1"/>
        <ExampColl Include="Item2"/>
        <ExampColl Include="Item3"/>
        <ExampColl Include="Item4"/>
        <ExampColl Include="Item5"/>
        <ExampColl Include="Item6"/>

    </ItemGroup>

    <Target Name="ShowMessage">
        <Message
            Text = "Identity: '%(Identity)' -- Items in ExampColl: @(ExampColl)"/>
    </Target>

</Project>
```

The Message task displays the following information:

```
Identity: 'Item1' -- Items in ExampColl: Item1
Identity: 'Item2' -- Items in ExampColl: Item2
Identity: 'Item3' -- Items in ExampColl: Item3
Identity: 'Item4' -- Items in ExampColl: Item4
Identity: 'Item5' -- Items in ExampColl: Item5
Identity: 'Item6' -- Items in ExampColl: Item6
```

Filter item lists

Batching can be used to filter out certain items from an item list before passing it to a task. For example, filtering on the `Extension` well-known item metadata value allows you to run a task on only files with a specific extension.

The following example shows how to divide an item list into batches based on item metadata, and then filter those batches when they are passed into a task. The `ExampColl` item list is divided into three batches based on the `Number` item metadata. The `Condition` attribute of the task specifies that only batches with a `Number` item

metadata value of `2` will be passed into the task

```
<Project
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <ItemGroup>

        <ExampColl Include="Item1">
            <Number>1</Number>
        </ExampColl>
        <ExampColl Include="Item2">
            <Number>2</Number>
        </ExampColl>
        <ExampColl Include="Item3">
            <Number>3</Number>
        </ExampColl>
        <ExampColl Include="Item4">
            <Number>1</Number>
        </ExampColl>
        <ExampColl Include="Item5">
            <Number>2</Number>
        </ExampColl>
        <ExampColl Include="Item6">
            <Number>3</Number>
        </ExampColl>

    </ItemGroup>

    <Target Name="Exec">
        <Message
            Text = "Items in ExampColl: @(ExampColl)"
            Condition="'%(Number)'=='2'"/>
    </Target>

</Project>
```

The Message task displays the following information:

```
Items in ExampColl: Item2;Item5
```

See also

- Well-known item metadata
- Item element (MSBuild)
- ItemMetadata element (MSBuild)
- Batching
- MSBuild concepts
- MSBuild reference

Item metadata in target batching

2/21/2019 • 2 minutes to read • Edit Online

MSBuild has the ability to perform dependency analysis on the inputs and outputs of a build target. If it is determined that the inputs or outputs of the target are up-to-date, the target will be skipped and the build will proceed. `Target` elements use the `Inputs` and `Outputs` attributes to specify the items to inspect during dependency analysis.

If a target contains a task that uses batched items as inputs or outputs, the `Target` element of the target should use batching in its `Inputs` or `Outputs` attributes to enable MSBuild to skip batches of items that are already up-to-date.

Batch targets

The following example contains an item list named `Res` that is divided into two batches based on the `Culture` item metadata. Each of these batches is passed into the `AL` task, which creates an output assembly for each batch. By using batching on the `Outputs` attribute of the `Target` element, MSBuild can determine if each of the individual batches is up-to-date before running the target. Without using target batching, both batches of items would be run by the task every time the target was executed.

```
<Project
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <ItemGroup>
        <Res Include="Strings.fr.resources">
            <Culture>fr</Culture>
        </Res>
        <Res Include="Strings.jp.resources">
            <Culture>jp</Culture>
        </Res>
        <Res Include="Menus.fr.resources">
            <Culture>fr</Culture>
        </Res>
        <Res Include="Dialogs.fr.resources">
            <Culture>fr</Culture>
        </Res>
        <Res Include="Dialogs.jp.resources">
            <Culture>jp</Culture>
        </Res>
        <Res Include="Menus.jp.resources">
            <Culture>jp</Culture>
        </Res>
    </ItemGroup>

    <Target Name="Build"
        Inputs="@(Res)"
        Outputs="%(Culture)\MyApp.resources.dll">

        <AL Resources="@(Res)"
            TargetType="library"
            OutputAssembly="%(Culture)\MyApp.resources.dll">

    </Target>

</Project>
```

See also

- How to: Build incrementally
- Batching
- Target element (MSBuild)
- Item metadata in task batching

MSBuild transforms

7/12/2019 • 2 minutes to read • Edit Online

A transform is a one-to-one conversion of one item list to another. In addition to enabling a project to convert item lists, a transform enables a target to identify a direct mapping between its inputs and outputs. This topic explains transforms and how MSBuild uses them to build projects more efficiently.

Transform modifiers

Transforms are not arbitrary, but are limited by special syntax in which all transform modifiers must be in the format %(<ItemMetaDataName>). Any item metadata can be used as a transform modifier. This includes the well-known item metadata that is assigned to every item when it is created. For a list of well-known item metadata, see Well-known item metadata.

In the following example, a list of *.resx* files is transformed into a list of *.resources* files. The %(filename) transform modifier specifies that each *.resources* file has the same file name as the corresponding *.resx* file.

```
@(RESXFile->'%(filename).resources')
```

For example, if the items in the @(RESXFile) item list are *Form1.resx*, *Form2.resx*, and *Form3.resx*, the outputs in the transformed list will be *Form1.resources*, *Form2.resources*, and *Form3.resources*.

> **NOTE**
> You can specify a custom separator for a transformed item list in the same way you specify a separator for a standard item list. For example, to separate a transformed item list by using a comma (,) instead of the default semicolon (;), use the following XML: `@(RESXFile->'Toolset\%(filename)%(extension)', ',')`

Use multiple modifiers

A transform expression can contain multiple modifiers, which can be combined in any order and can be repeated. In the following example, the name of the directory that contains the files is changed but the files retain the original name and file name extension.

```
@(RESXFile->'Toolset\%(filename)%(extension)')
```

For example, if the items that are contained in the `RESXFile` item list are *Project1\Form1.resx*, *Project1\Form2.resx*, and *Project1\Form3.text*, the outputs in the transformed list will be *Toolset\Form1.resx*, *Toolset\Form2.resx*, and *Toolset\Form3.text*.

Dependency analysis

Transforms guarantee a one-to-one mapping between the transformed item list and the original item list. Therefore, if a target creates outputs that are transforms of the inputs, MSBuild can analyze the timestamps of the inputs and outputs, and decide whether to skip, build, or partially rebuild a target.

In the Copy task in the following example, every file in the `BuiltAssemblies` item list maps to a file in the destination folder of the task, specified by using a transform in the `Outputs` attribute. If a file in the

`BuiltAssemblies` item list changes, the `Copy` task runs only for the changed file, and all other files are skipped. For more information about dependency analysis and how to use transforms, see How to: Build incrementally.

```
<Target Name="CopyOutputs"
    Inputs="@(BuiltAssemblies)"
    Outputs="@(BuiltAssemblies -> '$(OutputPath)%(Filename)%(Extension)')">

    <Copy
        SourceFiles="@(BuiltAssemblies)"
        DestinationFolder="$(OutputPath)"/>

</Target>
```

Example

Description

The following example shows an MSBuild project file that uses transforms. This example assumes that there is just one *.xsd* file in the *c:\sub0\sub1\sub2\sub3* directory, and that the working directory is *c:\sub0*.

Code

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <ItemGroup>
        <Schema Include="sub1\**\*.xsd"/>
    </ItemGroup>

    <Target Name="Messages">
        <Message Text="rootdir: @(Schema->'%(rootdir)')"/>
        <Message Text="fullpath: @(Schema->'%(fullpath)')"/>
        <Message Text="rootdir + directory + filename + extension: @(Schema->'%(rootdir)%(directory)%
(filename)%(extension)')"/>
        <Message Text="identity: @(Schema->'%(identity)')"/>
        <Message Text="filename: @(Schema->'%(filename)')"/>
        <Message Text="directory: @(Schema->'%(directory)')"/>
        <Message Text="relativedir: @(Schema->'%(relativedir)')"/>
        <Message Text="extension: @(Schema->'%(extension)')"/>
    </Target>
</Project>
```

Comments

This example produces the following output:

```
rootdir: C:\
fullpath: C:\sub0\sub1\sub2\sub3\myfile.xsd
rootdir + directory + filename + extension: C:\sub0\sub1\sub2\sub3\myfile.xsd
identity: sub1\sub2\sub3\myfile.xsd
filename: myfile
directory: sub0\sub1\sub2\sub3\
relativedir: sub1\sub2\sub3\
extension: .xsd
```

See also

- MSBuild concepts
- MSBuild reference
- How to: Build incrementally

Visual Studio integration (MSBuild)

9/11/2019 • 10 minutes to read • Edit Online

Visual Studio hosts MSBuild to load and build managed projects. Because MSBuild is responsible for the project, almost any project in the MSBuild format can be successfully used in Visual Studio, even if the project was authored by a different tool and has a customized build process.

This article describes specific aspects of Visual Studio's MSBuild hosting that should be considered when customizing projects and *.targets* files that you wish to load and build in Visual Studio. These will help you make sure Visual Studio features like IntelliSense and debugging work for your custom project.

For information about C++ projects, see Project files.

Project file name extensions

MSBuild.exe recognizes any project file name extension matching the pattern *.*proj*. However, Visual Studio only recognizes a subset of these project file name extensions, which determine the language-specific project system that will load the project. Visual Studio does not have a language-neutral MSBuild based project system.

For example, the Visual C# project system loads *.csproj* files, but Visual Studio is not able to load a *.xxproj* file. A project file for source files in an arbitrary language must use the same extension as Visual Basic or Visual C# project files to be loaded in Visual Studio.

Well-known target names

Clicking the **Build** command in Visual Studio will execute the default target in the project. Often, this target is also named `Build`. Choosing the **Rebuild** or **Clean** command will attempt to execute a target of the same name in the project. Clicking **Publish** will execute a target named `PublishOnly` in the project.

Configurations and platforms

Configurations are represented in MSBuild projects by properties grouped in a `PropertyGroup` element that contains a `Condition` attribute. Visual Studio looks at these conditions in order to create a list of project configurations and platforms to display. To successfully extract this list, the conditions must have a format similar to the following:

```
Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' "
Condition=" '$(Configuration)' == 'Release' "
Condition=" '$(Something)|$(Configuration)|$(SomethingElse)' == 'xxx|Debug|yyy' "
```

Visual Studio looks at the conditions on `PropertyGroup`, `ItemGroup`, `Import`, property, and item elements for this purpose.

Additional build actions

Visual Studio allows you to change the item type name of a file in a project with the **Build Action** property of the **File properties** window. **Compile**, **EmbeddedResource**, **Content**, and **None** item type names are always listed in this menu, along with any other item type names already in your project. To ensure any custom item type names are always available in this menu, you can add the names to an item type named `AvailableItemName`. For example, adding the following to your project file will add the custom type **JScript** to this menu for all projects that import

it:

```
<ItemGroup>
    <AvailableItemName Include="JScript"/>
</ItemGroup>
```

> **NOTE**
> Some item type names are special to Visual Studio but not listed in this dropdown.

In-process compilers

When possible, Visual Studio will attempt to use the in-process version of the Visual Basic compiler for increased performance. (Not applicable to Visual C#.) For this to work correctly, the following conditions must be met:

- In a target of the project, there must be a task named `Vbc` for Visual Basic projects.
- The `UseHostCompilerIfAvailable` parameter of the task must be set to true.

Design-time IntelliSense

To get IntelliSense support in Visual Studio before a build has generated an output assembly, the following conditions must be met:

- There must be a target named `Compile`.
- Either the `Compile` target or one of its dependencies must call the compiler task for the project, such as `Csc` or `Vbc`.
- Either the `Compile` target or one of its dependencies must cause the compiler to receive all the necessary parameters for IntelliSense, particularly all references.
- The conditions listed in the In-process compilers section must be met.

Build solutions

Within Visual Studio, the solution file and project build ordering are controlled by Visual Studio itself. When building a solution with *msbuild.exe* on the command line, MSBuild parses the solution file and orders the project builds. In both cases the projects are built individually in dependency order, and project to project references are not traversed. In contrast, when individual projects are built with *msbuild.exe*, project to project references are traversed.

When building inside Visual Studio, the property `$(BuildingInsideVisualStudio)` is set to `true`. This can be used in your project or *.targets* files to cause the build to behave differently.

Display properties and items

Visual Studio recognizes certain property names and values. For example, the following property in a project will cause **Windows Application** to appear in the **Application Type** box in the **Project Designer**.

```
<OutputType>WinExe</OutputType>
```

The property value can be edited in the **Project Designer** and saved in the project file. If such a property is given an invalid value by hand-editing, Visual Studio will show a warning when the project is loaded and replace the

invalid value with a default value.

Visual Studio understands defaults for some properties. These properties will not be persisted into the project file unless they have non-default values.

Properties with arbitrary names are not displayed in Visual Studio. To modify arbitrary properties in Visual Studio, you must open the project file in the XML editor and edit them by hand. For more information, see the Edit project files in Visual Studio section later in this topic.

Items defined in the project with arbitrary item type names are by default displayed in the **Solution Explorer** under their project node. To hide an item from display, set the `Visible` metadata to `false`. For example, the following item will participate in the build process but not be displayed in **Solution Explorer**.

```
<ItemGroup>
    <IntermediateFile Include="cache.temp">
        <Visible>false</Visible>
    </IntermediateFile>
</ItemGroup>
```

Items declared in files imported into the project are not displayed by default. Items created during the build process are never displayed in **Solution Explorer**.

Conditions on items and properties

During a build, all conditions are fully respected.

When determining property values to display, properties that Visual Studio considers configuration dependent are evaluated differently than properties it considers configuration independent. For properties it considers configuration dependent, Visual Studio sets the `Configuration` and `Platform` properties appropriately and instructs MSBuild to re-evaluate the project. For properties it considers configuration independent, it is indeterminate how conditions will be evaluated.

Conditional expressions on items are always ignored for the purposes of deciding whether the item should be displayed in **Solution Explorer**.

Debugging

In order to find and launch the output assembly and attach the debugger, Visual Studio needs the properties `OutputPath`, `AssemblyName`, and `OutputType` correctly defined. The debugger will fail to attach if the build process did not cause the compiler to generate a *.pdb* file.

Design-time target execution

Visual Studio attempts to execute targets with certain names when it loads a project. These targets include `Compile`, `ResolveAssemblyReferences`, `ResolveCOMReferences`, `GetFrameworkPaths`, and `CopyRunEnvironmentFiles`. Visual Studio runs these targets so that the compiler can be initialized to provide IntelliSense, the debugger can be initialized, and references displayed in Solution Explorer can be resolved. If these targets are not present, the project will load and build correctly but the design-time experience in Visual Studio will not be fully functional.

Edit project files in Visual Studio

To edit an MSBuild project directly, you can open the project file in the Visual Studio XML editor.

To unload and edit a project file in Visual Studio

1. In **Solution Explorer**, open the shortcut menu for the project, and then choose **Unload Project**.

The project is marked **(unavailable)**.

2. In **Solution Explorer**, open the shortcut menu for the unavailable project, and then choose **Edit <Project File>**.

   The project file opens in the Visual Studio XML Editor.

3. Edit, save, and then close the project file.

4. In **Solution Explorer**, open the shortcut menu for the unavailable project, and then choose **Reload Project**.

IntelliSense and validation

When using the XML editor to edit project files, IntelliSense and validation is driven by the MSBuild schema files. These are installed in the schema cache, which can be found in *<Visual Studio installation directory>\Xml\Schemas\1033\MSBuild*.

The core MSBuild types are defined in *Microsoft.Build.Core.xsd* and common types used by Visual Studio are defined in *Microsoft.Build.CommonTypes.xsd*. To customize the schemas so that you have IntelliSense and validation for custom item type names, properties, and tasks, you can either edit *Microsoft.Build.xsd*, or create your own schema that includes the CommonTypes or Core schemas. If you create your own schema you will have to direct the XML editor to find it using the **Properties** window.

Edit loaded project files

Visual Studio caches the content of project files and files imported by project files. If you edit a loaded project file, Visual Studio will automatically prompt you to reload the project so that the changes take effect. However if you edit a file imported by a loaded project, there will be no reload prompt and you must unload and reload the project manually to make the changes take effect.

Output groups

Several targets defined in *Microsoft.Common.targets* have names ending in `OutputGroups` or `OutputGroupDependencies`. Visual Studio calls these targets to get specific lists of project outputs. For example, the `SatelliteDllsProjectOutputGroup` target creates a list of all the satellite assemblies a build will create. These output groups are used by features like publishing, deployment, and project to project references. Projects that do not define them will load and build in Visual Studio, but some features may not work correctly.

Reference resolution

Reference resolution is the process of using the reference items stored in a project file to locate actual assemblies. Visual Studio must trigger reference resolution in order to show detailed properties for each reference in the **Properties** window. The following list describes the three types of references and how they are resolved.

- Assembly references:

  The project system calls a target with the well-known name `ResolveAssemblyReferences`. This target should produce items with the item type name `ReferencePath`. Each of these items should have an item specification (the value of the `Include` attribute of an item) containing the full path to the reference. The items should have all the metadata from the input items passed through in addition to the following new metadata:

  - `CopyLocal`, indicating whether the assembly should be copied into the output folder, set to true or false.

- `OriginalItemSpec`, containing the original item specification of the reference.
- `ResolvedFrom`, set to "{TargetFrameworkDirectory}" if it was resolved from the .NET Framework directory.

- COM references:

  The project system calls a target with the well-known name `ResolveCOMReferences`. This target should produce items with the item type name `ComReferenceWrappers`. Each of these items should have an item specification containing the full path to the interop assembly for the COM reference. The items should have all the metadata from the input items passed through, in addition to new metadata with the name `CopyLocal`, indicating whether the assembly should be copied into the output folder, set to true or false.

- Native references

  The project system calls a target with the well-known name `ResolveNativeReferences`. This target should produce items with the item type name `NativeReferenceFile`. The items should have all the metadata from the input items passed through, in addition to a new piece of metadata named `OriginalItemSpec`, containing the original item specification of the reference.

Performance shortcuts

If you use the Visual Studio IDE to start debugging (either by choosing the F5 key or by choosing **Debug** > **Start Debugging** on the menu bar) or to build your project (for example, **Build** > **Build Solution**), the build process uses a fast update check to improve performance. In some cases where customized builds create files that get built in turn, the fast update check does not correctly identify the changed files. Projects that need more thorough update checks can turn off the fast checking by setting the environment variable `DISABLEFASTUPTODATECHECK=1`. Alternatively, projects can set this as an MSBuild property in the project or in a file the project imports.

For regular builds in Visual Studio, the fast update check doesn't apply, and the project will build as if you invoked the build at a command prompt.

See also

- How to: Extend the Visual Studio build process
- Start a build from within the IDE
- Register extensions of the .NET Framework
- MSBuild concepts
- Item element (MSBuild)
- Property element (MSBuild)
- Target element (MSBuild)
- Csc task
- Vbc task

How to: Extend the Visual Studio build process

9/11/2019 • 4 minutes to read • Edit Online

The Visual Studio build process is defined by a series of MSBuild *.targets* files that are imported into your project file. One of these imported files, *Microsoft.Common.targets*, can be extended to allow you to run custom tasks at several points in the build process. This article explains two methods you can use to extend the Visual Studio build process:

- Overriding specific predefined targets defined in the common targets (*Microsoft.Common.targets* or the files that it imports).
- Overriding the "DependsOn" properties defined in the common targets.

Override predefined targets

The common targets contains a set of predefined empty targets that is called before and after some of the major targets in the build process. For example, MSBuild calls the `BeforeBuild` target before the main `CoreBuild` target and the `AfterBuild` target after the `CoreBuild` target. By default, the empty targets in the common targets do nothing, but you can override their default behavior by defining the targets you want in a project file that imports the common targets. By overriding the predefined targets, you can use MSBuild tasks to give you more control over the build process.

> **NOTE**
> SDK-style projects have an implicit import of targets *after the last line of the project file*. This means that you cannot override default targets unless you specify your imports manually as described in How to: Use MSBuild project SDKs.

To override a predefined target

1. Identify a predefined target in the common targets that you want to override. See the table below for the complete list of targets that you can safely override.
2. Define the target or targets at the end of your project file, immediately before the `</Project>` tag. For example:

```
<Project>
    ...
    <Target Name="BeforeBuild">
        <!-- Insert tasks to run before build here -->
    </Target>
    <Target Name="AfterBuild">
        <!-- Insert tasks to run after build here -->
    </Target>
</Project>
```

3. Build the project file.

The following table shows all of the targets in the common targets that you can safely override.

TARGET NAME	DESCRIPTION

TARGET NAME	DESCRIPTION
`BeforeCompile`, `AfterCompile`	Tasks that are inserted in one of these targets run before or after core compilation is done. Most customizations are done in one of these two targets.
`BeforeBuild`, `AfterBuild`	Tasks that are inserted in one of these targets will run before or after everything else in the build. **Note:** The `BeforeBuild` and `AfterBuild` targets are already defined in comments at the end of most project files, allowing you to easily add pre- and post-build events to your project file.
`BeforeRebuild`, `AfterRebuild`	Tasks that are inserted in one of these targets run before or after the core rebuild functionality is invoked. The order of target execution in *Microsoft.Common.targets* is: `BeforeRebuild`, `Clean`, `Build`, and then `AfterRebuild`.
`BeforeClean`, `AfterClean`	Tasks that are inserted in one of these targets run before or after the core clean functionality is invoked.
`BeforePublish`, `AfterPublish`	Tasks that are inserted in one of these targets run before or after the core publish functionality is invoked.
`BeforeResolveReferences`, `AfterResolveReferences`	Tasks that are inserted in one of these targets run before or after assembly references are resolved.
`BeforeResGen`, `AfterResGen`	Tasks that are inserted in one of these targets run before or after resources are generated.

Override DependsOn properties

Overriding predefined targets is an easy way to extend the build process, but, because MSBuild evaluates the definition of targets sequentially, there is no way to prevent another project that imports your project from overriding the targets you already have overridden. So, for example, the last `AfterBuild` target defined in the project file, after all other projects have been imported, will be the one that is used during the build.

You can guard against unintended overrides of targets by overriding the DependsOn properties that are used in `DependsOnTargets` attributes throughout the common targets. For example, the `Build` target contains a `DependsOnTargets` attribute value of `"$(BuildDependsOn)"`. Consider:

```
<Target Name="Build" DependsOnTargets="$(BuildDependsOn)"/>
```

This piece of XML indicates that before the `Build` target can run, all the targets specified in the `BuildDependsOn` property must run first. The `BuildDependsOn` property is defined as:

```
<PropertyGroup>
    <BuildDependsOn>
        BeforeBuild;
        CoreBuild;
        AfterBuild
    </BuildDependsOn>
</PropertyGroup>
```

You can override this property value by declaring another property named `BuildDependsOn` at the end of your project file. By including the previous `BuildDependsOn` property in the new property, you can add new targets to the

beginning and end of the target list. For example:

```
<PropertyGroup>
    <BuildDependsOn>
        MyCustomTarget1;
        $(BuildDependsOn);
        MyCustomTarget2
    </BuildDependsOn>
</PropertyGroup>

<Target Name="MyCustomTarget1">
    <Message Text="Running MyCustomTarget1..."/>
</Target>
<Target Name="MyCustomTarget2">
    <Message Text="Running MyCustomTarget2..."/>
</Target>
```

Projects that import your project files can override these properties without overwriting the customizations that you have made.

To override a DependsOn property

1. Identify a predefined DependsOn property in the common targets that you want to override. See the table below for a list of the commonly overridden DependsOn properties.
2. Define another instance of the property or properties at the end of your project file. Include the original property, for example `$(BuildDependsOn)`, in the new property.
3. Define your custom targets before or after the property definition.
4. Build the project file.

Commonly overridden DependsOn properties

PROPERTY NAME	DESCRIPTION
`BuildDependsOn`	The property to override if you want to insert custom targets before or after the entire build process.
`CleanDependsOn`	The property to override if you want to clean up output from your custom build process.
`CompileDependsOn`	The property to override if you want to insert custom processes before or after the compilation step.

See also

- Visual Studio integration
- MSBuild concepts
- .targets files

Start a build from within the IDE

4/15/2019 • 2 minutes to read • Edit Online

Custom project systems must use IVsBuildManagerAccessor to start builds. This article describes the reasons for this requirement and outlines the procedure.

Parallel builds and threads

Visual Studio allows parallel builds, which requires mediation for access to common resources. Project systems can run builds asynchronously, but such systems must not call build functions from within call-backs.

If the project system modifies environment variables, it must set the NodeAffinity of the build to OutOfProc. This requirement means that you cannot use host objects, since they require the in-proc node.

Use IVSBuildManagerAccessor

The code below outlines a method that a project system can use to start a build:

```
public bool Build(Project project, bool isDesignTimeBuild)
{
    // Get the accessor from the IServiceProvider interface for the
    // project system
    IVsBuildManagerAccessor accessor =
        serviceProvider.GetService(typeof(SVsBuildManagerAccessor)) as
        IVsBuildManagerAccessor;
    bool releaseUIThread = false;
    try
    {
        if(accessor != null)
        {
            // Claim the UI thread under the following conditions:
            // 1. The build must use a resource that uses the UI thread
            // or,
            // 2. The build requires the in-proc node AND waits on the
            // UI thread for the build to complete
            if(NeedsUIThread)
            {
                int result = accessor.ClaimUIThreadForBuild();
                if(result != S_OK)
                {
                     // Not allowed to claim the UI thread right now
                     return false;
                }
                releaseUIThread = true;
             }
             if(isDesignTimeBuild)
             {
// Start the design time build
                  int result = accessor.BeginDesignTimeBuild();
                  if(result != S_OK)
                  {
                      // Not allowed to begin a design-time build at
                      // this time. Try again later.
                      return false;
                  }
             }
         }
         bool buildSucceeded = false;
```

```
        // perform project-system specific build set up tasks
        // Create your BuildRequestData
        // This assumes a IHostServices variable (hostServices) set
   // to your host services. If you don't use a project instance
        // (you build from a file for example) then use another
        // constructor.
        BuildRequestData requestData = new
            BuildRequestData(project.CreateProjectInstance(),
            "myTarget", hostServices,
            BuildRequestData.BuildRequestDataFlags.None);
        // Mark your your submission as Pending
        BuildSubmission submission =
             BuildManager.DefaultBuildManager.
             PendBuildRequest(requestData);
        // Register the loggers in BuildLoggers
        if (accessor != null)
        {
              foreach (ILogger logger in BuildLoggers)
              {
                   accessor.RegisterLogger(submission.SubmissionId,
                       logger);
              }
        }
        BuildResult buildResult = submission.Execute();
        return buildResult;
    }
    // Clean up resources
    finally
    {
        if(accessor != null)
        {
            // Unregister the loggers, if necessary.
            accessor.UnregisterLoggers(submission.SubmissionId);
            // Release the UI thread, if used
            if(releaseUIThread)
            {
                 accessor.ReleaseUIThreadForBuild();
            }
            // End the design time build, if used
            if(isDesignTimeBuild)
            {
                 accessor.EndDesignTimeBuild();
            }
        }
    }
}
```

Register extensions of the .NET Framework

4/23/2019 • 2 minutes to read • Edit Online

You can develop an assembly that extends a specific version of the .NET Framework. To enable the assembly to appear in the Visual Studio **Add References** dialog box, you must add the folder that contains it to the system registry.

For example, assume that the Trey Research company has developed a library that extends the .NET Framework 4, and wants the library assemblies to appear in the **Add References** dialog box when a project targets the .NET Framework 4. Also assume that the assemblies are 32-bit assemblies running on a 32-bit computer or 64-bit assemblies running on a 64-bit computer, and that they will be installed in the *C:\TreyResearch\Extensions4* folder.

Register this folder by using this key: **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\.NETFramework\v4.0.21006\AssemblyFoldersEx\Trey Research**. Give the key this default value: **C:\TreyResearch\Extensions4**.

> **NOTE**
> The build number of the .NET Framework version may be different.

To register a 32-bit assembly on a 64-bit computer, use the Wow6432 node, for example: **HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\.NETFramework\v4.0.21006\AssemblyFoldersEx\TreyResearch**.

See also

- Visual Studio integration

Build multiple projects in parallel with MSBuild

10/24/2019 • 2 minutes to read • Edit Online

You can use MSBuild to build multiple projects faster by running them in parallel. To run builds in parallel, you use the following settings on a multi-core or multiple processor computer:

- The `-maxcpucount` switch at a command prompt.
- The BuildInParallel task parameter on an MSBuild task.

> **NOTE**
> The **-verbosity** (**-v**) switch in a command line can also affect build performance. Your build performance might decrease if the verbosity of your build log information is set to detailed or diagnostic, which are used for troubleshooting. For more information, see Obtain build logs and Command-line reference.

-maxcpucount Switch

If you use the `-maxcpucount` switch, or `-m` for short, MSBuild can create the specified number of *MSBuild.exe* processes that may be run in parallel. These processes are also known as "worker processes." Each worker process uses a separate core or processor, if any are available, to build a project at the same time as other available processors may be building other projects. For example, setting this switch to a value of "4" causes MSBuild to create four worker processes to build the project.

If you include the `-maxcpucount` switch without specifying a value, MSBuild will use up to the number of processors on the computer.

For more information about this switch, which was introduced in MSBuild 3.5, see Command-line reference.

The following example instructs MSBuild to use three worker processes. If you use this configuration, MSBuild can build three projects at the same time.

```
msbuild.exe myproj.proj -maxcpucount:3
```

BuildInParallel task parameter

`BuildInParallel` is an optional boolean parameter on a MSBuild task. When `BuildInParallel` is set to `true` (its default value is `false`), multiple worker processes are generated to build as many projects at the same time as possible. For this to work correctly, the `-maxcpucount` switch must be set to a value greater than 1, and the system must be at least dual-core or have two or more processors.

The following is an example, taken from *microsoft.common.targets*, about how to set the `BuildInParallel` parameter.

```
<PropertyGroup>
    <BuildInParallel Condition="'$(BuildInParallel)' ==
        ''">true</BuildInParallel>
</PropertyGroup>
<MSBuild
    Projects="@(_MSBuildProjectReferenceExistent)"
    Targets="GetTargetPath"
    BuildInParallel="$(BuildInParallel)"
    Properties="%(_MSBuildProjectReferenceExistent.SetConfiguration);
        %(_MSBuildProjectReferenceExistent.SetPlatform)"
    Condition="'@(NonVCProjectReference)'!='' and
        ('$(BuildingSolutionFile)' == 'true' or
        '$(BuildingInsideVisualStudio)' == 'true' or
        '$(BuildProjectReferences)' != 'true') and
        '@(_MSBuildProjectReferenceExistent)' != ''"
    ContinueOnError="!$(BuildingProject)">
    <Output TaskParameter="TargetOutputs"
        ItemName="_ResolvedProjectReferencePaths"/>
</MSBuild>
```

See also

- Use multiple processors to build projects
- Write multi-processor-aware loggers
- Tuning C++ build parallelism blog

Use multiple processors to build projects

10/21/2019 • 2 minutes to read • Edit Online

MSBuild can take advantage of systems that have multiple processors, or multiple-core processors. A separate build process is created for each available processor. For example, if the system has four processors, then four build processes are created. MSBuild can process these builds simultaneously, and therefore overall build time is reduced. However, parallel building introduces some changes in how build processes occur. This topic discusses those changes.

Project-to-project references

When the Microsoft Build Engine encounters a project-to-project (P2P) reference while it is using parallel builds to build a project, it builds the reference only one time. If two projects have the same P2P reference, the reference is not rebuilt for each project. Instead, the build engine returns the same P2P reference to both projects that depend on it. Future requests in the session for the same target are provided the same P2P reference.

Cycle detection

Cycle detection functions the same as it did in MSBuild 2.0, except that now MSBuild can report the detection of the cycle at a different time or in the build.

Errors and exceptions during parallel builds

In parallel builds, errors and exceptions can occur at different times than they do in a non-parallel build, and when one project does not build, the other project builds continue. MSBuild will not stop any project build that is building in parallel with the one that failed. Other projects continue to build until they either succeed or fail. However, if ContinueOnError has been enabled, then no builds will stop even if an error occurs.

C++ project (.vcxproj) and solution (.sln) files

Both Visual C++ projects (*.vcxproj*) and solution (*.sln*) files can be passed to the MSBuild task. For Visual C++ projects, VCWrapperProject is called, and then the internal MSBuild project is created. For Visual C++ solutions, a SolutionWrapperProject is created, and then the internal MSBuild project is created. In both cases, the resulting project is treated the same as any other MSBuild project.

Multi-process execution

Almost all build-related activities require the current directory to remain constant throughout the build process to prevent path-related errors. Therefore, projects cannot run on different threads in MSBuild because they would cause multiple directories to be created.

To avoid this problem but still enable multi-processor builds, MSBuild uses "process isolation." By using process isolation, MSBuild can create a maximum of `n` processes, where `n` equals the number of processors available on the system. For example, if MSBuild builds a solution on a system that has two processors, then only two build processes are created. These processes are re-used to build all the projects in the solution.

See also

- Build multiple projects in parallel
- Tasks

Use memory efficiently when you build large projects

2/21/2019 • 2 minutes to read • Edit Online

Large projects often contain many subprojects and other dependencies, which may consume lots of system memory at build time. When available system memory is decreased, system performance may also be decreased. Older versions of MSBuild projects remained in memory. Version 3.5 removed older versions of projects, but retained build results in a cache for later retrieval.

Version 4.0 handles this memory management automatically, saving projects from having to use properties such as `UnloadProjectsOnCompletion` and `UseResultsCache`.

See also

- Build multiple projects in parallel

MSBuild multitargeting overview

8/9/2019 • 2 minutes to read • Edit Online

By using MSBuild, you can compile an application to run on any one of several versions of the .NET Framework, and on any one of several system platforms. For example, you can compile an application to run on the .NET Framework 2.0 on a 32-bit platform, and compile the same application to run on the .NET Framework 4.5 on a 64-bit platform.

> **IMPORTANT**
> Despite the name "multitargeting", a project can target only one framework and only one platform at a time.

These are some of the features of MSBuild targeting:

- You can develop an application that targets an earlier version of the .NET Framework, for example, versions 2.0, 3.5, or 4.
- You can target a framework other than the .NET Framework, for example, the Silverlight Framework.
- You can target a *framework profile*, which is a predefined subset of a target framework.
- If a service pack for the current version of the .NET Framework is released, you could target it.
- MSBuild targeting guarantees that an application uses only the functionality that is available in the targeted framework and platform.

Target framework and platform

A *target framework* is the version of the .NET Framework that a project is built to run on, and a *target platform* is the system platform that the project is built to run on. For example, you might want to target a .NET Framework 2.0 application to run on a 32-bit platform that is compatible with the 802x86 processor family (x86). The combination of target framework and target platform is known as the *target context*. For more information, see Target framework and target platform.

Toolset (ToolsVersion)

A Toolset collects together the tools, tasks, and targets that are used to create the application. A Toolset includes compilers such as *csc.exe* and *vbc.exe*, the common targets file (*microsoft.common.targets*), and the common tasks file (*microsoft.common.tasks*). The 4.5 Toolset can be used to target .NET Framework versions 2.0, 3.0, 3.5, 4, and 4.5. However, the 2.0 Toolset can only be used to target the .NET Framework version 2.0. For more information, see Toolset (ToolsVersion).

Reference assemblies

The reference assemblies that are specified in the Toolset help you design and build an application. These reference assemblies not only enable a particular target build, but also restrict components and features in the Visual Studio IDE to those that are compatible with the target. For more information, see Resolve assemblies at design time.

Configure targets and tasks

You can configure MSBuild targets and tasks to run out-of-process with MSBuild so that you can target contexts that are considerably different than the one you are running on. For example, you can target a 32-bit, .NET Framework 2.0 application while the development computer is running on a 64-bit platform with .NET Framework 4.5. For more information, see Configure targets and tasks.

Troubleshooting

You might encounter errors if you try to reference an assembly that is not part of the target context. For more information about these errors and what to do about them, see Troubleshoot .NET Framework targeting errors.

MSBuild Toolset (ToolsVersion)

8/9/2019 • 5 minutes to read • Edit Online

MSBuild uses a Toolset of tasks, targets, and tools to build an application. Typically, a MSBuild Toolset includes a *microsoft.common.tasks* file, a *microsoft.common.targets* file, and compilers such as *csc.exe* and *vbc.exe*. Most Toolsets can be used to compile applications to more than one version of the .NET Framework and more than one system platform. However, the MSBuild 2.0 Toolset can be used to target only the .NET Framework 2.0.

ToolsVersion attribute

Specify the Toolset in the `ToolsVersion` attribute on the Project element in the project file. The following example specifies that the project should be built by using the MSBuild "Current" Toolset.

```
<Project ToolsVersion="Current" ... </Project>
```

Specify the Toolset in the `ToolsVersion` attribute on the Project element in the project file. The following example specifies that the project should be built by using the MSBuild 15.0 Toolset.

```
<Project ToolsVersion="15.0" ... </Project>
```

> **NOTE**
> Some project types use the `sdk` attribute instead of `ToolsVersion`. For more information, see Packages, metadata, and frameworks and Additions to the csproj format for .NET Core.

How the ToolsVersion attribute works

When you create a project in Visual Studio, or upgrade an existing project, an attribute named `ToolsVersion` is automatically included in the project file and its value corresponds to the version of MSBuild that is included in the Visual Studio edition. For more information, see Framework targeting overview.

When a `ToolsVersion` value is defined in a project file, MSBuild uses that value to determine the values of the Toolset properties that are available to the project. One Toolset property is `$(MSBuildToolsPath)`, which specifies the path of the .NET Framework tools. Only that Toolset property (or `$(MSBuildBinPath)`), is required.

Starting in Visual Studio 2013, the MSBuild Toolset version is the same as the Visual Studio version number. MSBuild defaults to this Toolset within Visual Studio and on the command line, regardless of the Toolset version specified in the project file. This behavior can be overridden by using the -ToolsVersion flag. For more information, see Override ToolsVersion settings.

In the following example, MSBuild finds the *Microsoft.CSharp.targets* file by using the `MSBuildToolsPath` reserved property.

```
<Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />
```

You can modify the value of `MSBuildToolsPath` by defining a custom Toolset. For more information, see Standard and custom Toolset configurations.

When you build a solution on the command line and specify a `ToolsVersion` for *msbuild.exe*, all projects and their project-to-project dependencies are built according to that `ToolsVersion`, even if each project in the solution specifies its own `ToolsVersion`. To define the `ToolsVersion` value on a per project basis, see Overriding ToolsVersion settings.

The `ToolsVersion` attribute is also used for project migration. For example, if you open a Visual Studio 2008 project in Visual Studio 2010, the project file is updated to include ToolsVersion="4.0". If you then try to open that project in Visual Studio 2008, it doesn't recognize the upgraded `ToolsVersion` and therefore builds the project as though the attribute was still set to 3.5.

Visual Studio 2010 and Visual Studio 2012 use a ToolsVersion of 4.0. Visual Studio 2013 uses a ToolsVersion of 12.0. Visual Studio 2015 uses ToolsVersion 14.0, and Visual Studio 2017 uses ToolsVersion 15.0. In many cases, you can open the project in multiple versions of Visual Studio without modification. Visual Studio always uses the correct Toolset, but you will be notified if the version used does not match the version in the project file. In almost all cases, this warning is benign as the Toolsets are compatible in most cases.

Sub-toolsets, which are described later in this topic, allow MSBuild to automatically switch which set of tools to use based on the context in which the build is being run. For example, MSBuild uses a newer set of tools when it's run in Visual Studio 2012 than when it's run in Visual Studio 2010, without your having to explicitly change the project file.

Toolset implementation

Implement a Toolset by selecting the paths of the various tools, targets, and tasks that make up the Toolset. The tools in the Toolset that MSBuild defines come from the following sources:

- The .NET Framework folder.
- Additional managed tools.

  The managed tools include *ResGen.exe* and *TlbImp.exe*.

MSBuild provides two ways to access the Toolset:

- By using Toolset properties
- By using ToolLocationHelper methods

Toolset properties specify the paths of the tools. Starting in Visual Studio 2017, MSBuild no longer has a fixed location. By default, it is located in the *MSBuild\15.0\Bin* folder relative to the Visual Studio installation location. In earlier versions, MSBuild uses the value of the `ToolsVersion` attribute in the project file to locate the corresponding registry key, and then uses the information in the registry key to set the Toolset properties. For example, if `ToolsVersion` has the value `12.0`, then MSBuild sets the Toolset properties according to this registry key: **HKLM\Software\Microsoft\MSBuild\ToolsVersions\12.0**.

These are Toolset properties:

- `MSBuildToolsPath` specifies the path of the MSBuild binaries.
- `SDK40ToolsPath` specifies the path of additional managed tools for MSBuild 4.x (which could be 4.0 or 4.5).
- `SDK35ToolsPath` specifies the path of additional managed tools for MSBuild 3.5.

Alternately, you can determine the Toolset programmatically by calling the methods of the ToolLocationHelper class. The class includes these methods:

- GetPathToDotNetFramework returns the path of the .NET Framework folder.
- GetPathToDotNetFrameworkFile returns the path of a file in the .NET Framework folder.

- GetPathToDotNetFrameworkSdk returns the path of the managed tools folder.
- GetPathToDotNetFrameworkSdkFile returns the path of a file, which is typically located in the managed tools folder.
- GetPathToBuildTools returns the path of the build tools.

Sub-toolsets

For versions MSBuild prior to 15.0, MSBuild uses a registry key to specify the path of the basic tools. If the key has a subkey, MSBuild uses it to specify the path of a sub-toolset that contains additional tools. In this case, the Toolset is defined by combining the property definitions that are defined in both keys.

> **NOTE**
> If Toolset property names collide, the value that's defined for the subkey path overrides the value that's defined for the root key path.

Sub-toolsets become active in the presence of the `VisualStudioVersion` build property. This property may take one of these values:

- "10.0" specifies the .NET Framework 4 sub-toolset
- "11.0" specifies the .NET Framework 4.5 sub-toolset
- "12.0" specifies the .NET Framework 4.5.1 sub-toolset

Sub-toolsets 10.0 and 11.0 should be used with ToolsVersion 4.0. In later versions, the sub-toolset version and the ToolsVersion should match.

During a build, MSBuild automatically determines and sets a default value for the `VisualStudioVersion` property if it's not already defined.

MSBuild provides overloads for the `ToolLocationHelper` methods that add a `VisualStudioVersion` enumerated value as a parameter

Sub-toolsets were introduced in the .NET Framework 4.5.

See also

- Standard and custom Toolset configurations
- Multitargeting

Standard and custom Toolset configurations

4/23/2019 • 4 minutes to read • Edit Online

An MSBuild Toolset contains references to tasks, targets, and tools that you can use to build an application project. MSBuild includes a standard Toolset, but you can also create custom Toolsets. For information about how to specify a Toolset, see Toolset (ToolsVersion)

Standard Toolset configurations

MSBuild 16.0 includes the following standard Toolsets:

TOOLSVERSION	TOOLSET PATH (AS SPECIFIED IN THE MSBUILDTOOLSPATH OR MSBUILDBINPATH BUILD PROPERTY)
2.0	*<Windows installation path>\Microsoft.Net\Framework\v2.0.50727*
3.5	*<Windows installation path>\Microsoft.NET\Framework\v3.5*
4.0	*<Windows installation path>\Microsoft.NET\Framework\v4.0.30319*
Current	*<Visual Studio installation path>\MSBuild\Current\bin*

The `ToolsVersion` value determines which Toolset is used by a project that Visual Studio generates. In Visual Studio 2019, the default value is "Current" (no matter what the version specified in the project file), but you can override that attribute by using the **/toolsversion** switch at a command prompt. For information about this attribute and other ways to specify the `ToolsVersion`, see Overriding ToolsVersion settings.

MSBuild 15.0 includes the following standard Toolsets:

TOOLSVERSION	TOOLSET PATH (AS SPECIFIED IN THE MSBUILDTOOLSPATH OR MSBUILDBINPATH BUILD PROPERTY)
2.0	*<Windows installation path>\Microsoft.Net\Framework\v2.0.50727*
3.5	*<Windows installation path>\Microsoft.NET\Framework\v3.5*
4.0	*<Windows installation path>\Microsoft.NET\Framework\v4.0.30319*
15.0	*<Visual Studio installation path>\MSBuild\15.0\bin*

The `ToolsVersion` value determines which Toolset is used by a project that Visual Studio generates. In Visual Studio 2017, the default value is "15.0" (no matter what the version specified in the project file), but you can override that attribute by using the **/toolsversion** switch at a command prompt. For information about this attribute and other ways to specify the `ToolsVersion`, see Overriding ToolsVersion settings.

Visual Studio 2017 and later versions do not use a registry key for the path to MSBuild. For versions of MSBuild

prior to 15.0 that are installed with Visual Studio 2017, the following registry keys specify the installation path of MSBuild.exe.

REGISTRY KEY	KEY NAME	STRING KEY VALUE
\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSBuild\ToolsVersions\2.0	**MSBuildToolsPath**	**.NET Framework 2.0 Install Path**
\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSBuild\ToolsVersions\3.5	**MSBuildToolsPath**	**.NET Framework 3.5 Install Path**
\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSBuild\ToolsVersions\4.0	**MSBuildToolsPath**	**.NET Framework 4 Install Path**

Sub-toolsets

If the registry key in the previous table has a subkey, MSBuild uses it to determine the path of a sub-toolset that overrides the path in the parent Toolset. The following subkey is an example:

\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSBuild\ToolsVersions\12.0\12.0

If any properties are defined in both the base Toolset and the selected sub-toolset, the property definitions in the sub-toolset are used. For example, the MSBuild 4.0 Toolset defines `SDK40ToolsPath` to point to the 7.0A SDK, but the MSBuild 4.0\11.0 Toolset defines the same property to point to the 8.0A SDK. If `VisualStudioVersion` is unset, `SDK40ToolsPath` would point to 7.0A, but if `VisualStudioVersion` is set to 11.0, the property would instead point to 8.0A.

The `VisualStudioVersion` build property indicates whether a sub-toolset becomes active. For example, a `VisualStudioVersion` value of "12.0" specifies the MSBuild 12.0 sub-toolset. For more information, see the Sub-toolsets section of Toolset (ToolsVersion).

> **NOTE**
> We recommend that you avoid changing these settings. Nevertheless, you can add your own settings and define computer-wide custom Toolset definitions, as the next section describes.

Custom Toolset definitions

When a standard Toolset does not fulfill your build requirements, you can create a custom Toolset. For example, you may have a build lab scenario in which you must have a separate system for building Visual C++ projects. By using a custom Toolset, you can assign custom values to the `ToolsVersion` attribute when you create projects or run *MSBuild.exe*. By doing this, you can also use the `$(MSBuildToolsPath)` property to import *.targets* files from that directory, as well as defining your own custom Toolset properties that can be used for any project that uses that Toolset.

Specify a custom Toolset in the configuration file for *MSBuild.exe* (or for the custom tool that hosts the MSBuild engine if that is what you are using). For example, the configuration file for *MSBuild.exe* could include the following Toolset definition if you wished to define a toolset named *MyCustomToolset*.

```
<msbuildToolsets default="MyCustomToolset">
   <toolset toolsVersion="MyCustomToolset">
      <property name="MSBuildToolsPath"
        value="C:\SpecialPath" />
   </toolset>
</msbuildToolsets>
```

`<msbuildToolsets>` must also be defined in the configuration file, as follows.

```
<configSections>
   <section name="msbuildToolsets"
       Type="Microsoft.Build.BuildEngine.ToolsetConfigurationSection,
       Microsoft.Build, Version=15.1.0.0, Culture=neutral,
       PublicKeyToken=b03f5f7f11d50a3a"
   </section>
</configSections>
```

> **NOTE**
> To be read correctly, `<configSections>` must be the first subsection in the `<configuration>` section.

`ToolsetConfigurationSection` is a custom configuration section that can be used by any MSBuild host for custom configuration. If you use a custom Toolset, a host does not have to do anything to initialize the build engine except provide the configuration file entries. By defining entries in the registry, you can specify computer-wide Toolsets that apply to *MSBuild.exe*, Visual Studio, and all hosts of MSBuild.

> **NOTE**
> If a configuration file defines settings for a `ToolsVersion` that was already defined in the registry, the two definitions are not merged. The definition in the configuration file takes precedence and the settings in the registry for that `ToolsVersion` are ignored.

The following properties are specific to the value of `ToolsVersion` that is used in projects:

- **$(MSBuildBinPath)** is set to the `ToolsPath` value that is specified either in the registry or in the configuration file where the `ToolsVersion` is defined. The `$(MSBuildToolsPath)` setting in the registry or the configuration file specifies the location of the core tasks and targets. In the project file, this maps to the $(MSBuildBinPath) property, and also to the $(MSBuildToolsPath) property.

- `$(MSBuildToolsPath)` is a reserved property that is supplied by the MSBuildToolsPath property that is specified in the configuration file. (This property replaces `$(MSBuildBinPath)`. However, `$(MSBuildBinPath)` is carried forward for compatibility.) A custom Toolset must define either `$(MSBuildToolsPath)` or `$(MSBuildBinPath)` but not both, unless they both have the same value.

  You can also add custom, ToolsVersion-specific properties to the configuration file by using the same syntax that you use to add the MSBuildToolsPath property. To make these custom properties available to the project file, use the same name as the name of the value that is specified in the configuration file. You may define Toolsets but not sub-toolsets in the configuration file.

See also

- Toolset (ToolsVersion)

Override ToolsVersion settings

4/18/2019 • 2 minutes to read • Edit Online

You can change the Toolset for projects and solutions in one of three ways:

1. By using the `-ToolsVersion` switch (or `-tv`, for short) when you build the project or solution from the command line.
2. By setting the `ToolsVersion` parameter on the MSBuild task.
3. By setting the `$(ProjectToolsVersion)` property on a project within a solution. This lets you build a project in a solution with a Toolset version that differs from that of the other projects.

Override the ToolsVersion settings of projects and solutions on command line builds

Although Visual Studio projects typically build with the ToolsVersion specified in the project file, you can use the `-ToolsVersion` (or `-tv`) switch on the command line to override that value and build all of the projects and their project-to-project dependencies with a different Toolset. For example:

```
msbuild.exe someproj.proj -tv:12.0 -p:Configuration=Debug
```

In this example, all projects are built using ToolsVersion 12.0. (However, see the section Order of precedence later in this topic.)

When using the `-tv` switch on the command line, you can optionally use the `$(ProjectToolsVersion)` property in individual projects to build them with a different ToolsVersion value than the other projects in the solution.

Override the ToolsVersion settings using the ToolsVersion parameter of the MSBuild task

The MSBuild task is the primary means for one project to build another. To enable the MSBuild task to build a project with a different ToolsVersion than the one specified in the project, it provides an optional task parameter named `ToolsVersion`. The following example demonstrates how to use this parameter:

1. Create a file that's named *projectA.proj* and that contains the following code:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
ToolsVersion="12.0">

    <Target Name="go" >
        <Message Text="projectA.proj" />
        <Message Text="MSBuildToolsVersion: $(MSBuildToolsVersion)" />
        <Message Text="MSBuildToolsPath:    $(MSBuildToolsPath)" />

        <MSBuild Projects="projectB.proj"
            ToolsVersion="2.0"
            Targets="go" />
    </Target>
</Project>
```

2. Create another file that's named *projectB.proj* and that contains the following code:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
ToolsVersion="12.0">

    <Target Name="go">
        <Message Text="projectB.proj" />
        <Message Text="MSBuildToolsVersion: $(MSBuildToolsVersion)" />
        <Message Text="MSBuildToolsPath:    $(MSBuildToolsPath)" />
    </Target>
</Project>
```

3. Enter the following command at a command prompt:

```
msbuild projectA.proj -t:go -toolsversion:3.5
```

4. The following output appears. For `projectA`, the `-toolsversion:3.5` setting on the command line overrides the `ToolsVersion=12.0` setting in the `Project` tag.

   `ProjectB` is called by a task in `projectA`. That task has `ToolsVersion=2.0`, which overrides the other `ToolsVersion` settings for `projectB`.

```
Output:
  projectA.proj
  MSBuildToolsVersion: 3.5
  MSBuildToolsPath:    C:\Windows\Microsoft.NET\Framework\v3.5

  projectB.proj
  MSBuildToolsVersion: 2.0
  MSBuildToolsPath:    C:\Windows\Microsoft.NET\Framework\v2.0.50727
```

Order of precedence

The order of precedence, from highest to lowest, used to determine the `ToolsVersion` is:

1. The `ToolsVersion` attribute on the MSBuild task used to build the project, if any.
2. The `-toolsversion` (or `-tv`) switch that's used in the msbuild.exe command, if any.
3. If the environment variable `MSBUILDTREATALLTOOLSVERSIONSASCURRENT` is set, then use the current `ToolsVersion`.
4. If the environment variable `MSBUILDTREATHIGHERTOOLSVERSIONASCURRENT` is set and the `ToolsVersion` defined in the project file is greater than the current `ToolsVersion`, use the current `ToolsVersion`.
5. If the environment variable `MSBUILDLEGACYDEFAULTTOOLSVERSION` is set, or if `ToolsVersion` is not set, then the following steps are used:
   a. The `ToolsVersion` attribute of the Project element of the project file. If this attribute doesn't exist, it is assumed to be the current version.
   b. The default tools version in the *MSBuild.exe.config* file.
   c. The default tools version in the registry. For more information, see Standard and custom Toolset configurations.
6. If the environment variable `MSBUILDLEGACYDEFAULTTOOLSVERSION` is not set, then the following steps are used:
   a. If the environment variable `MSBUILDDEFAULTTOOLSVERSION` is set to a `ToolsVersion` that exists, use it.

b. If `DefaultOverrideToolsVersion` is set in *MSBuild.exe.config*, use it.

c. If `DefaultOverrideToolsVersion` is set in the registry, use it.

d. Otherwise, use the current `ToolsVersion`.

See also

- Multitargeting
- MSBuild concepts
- Toolset (ToolsVersion)
- Standard and custom Toolset configurations

MSBuild target framework and target platform

10/31/2019 • 2 minutes to read • Edit Online

A project can be built to run on a *target framework*, which is a particular version of the .NET Framework, and a *target platform*, which is a particular software architecture. For example, you can target an application to run on the .NET Framework 2.0 on a 32-bit platform that is compatible with the 802x86 processor family ("x86"). The combination of target framework and target platform is known as the *target context*.

> **IMPORTANT**
> This article shows the old way to specify a target framework. SDK-style projects enable different TargetFrameworks like netstandard. For more info, see Target frameworks.

Target framework and profile

A target framework is the particular version of the .NET Framework that your project is built to run on. Specification of a target framework is required because it enables compiler features and assembly references that are exclusive to that version of the framework.

Currently, the following versions of the .NET Framework are available for use:

- The .NET Framework 2.0 (included in Visual Studio 2005)
- The .NET Framework 3.0 (included in Windows Vista)
- The .NET Framework 3.5 (included in Visual Studio 2008)
- The .NET Framework 4.5.2
- The .NET Framework 4.6 (included in Visual Studio 2015)
- The .NET Framework 4.6.1
- The .NET Framework 4.6.2
- The .NET Framework 4.7
- The .NET Framework 4.7.1
- The .NET Framework 4.7.2
- The .NET Framework 4.8

The versions of the .NET Framework differ from one another in the list of assemblies that each makes available to reference. For example, you cannot build Windows Presentation Foundation (WPF) applications unless your project targets the .NET Framework version 3.0 or above.

The target framework is specified in the `TargetFrameworkVersion` property in the project file. You can change the target framework for a project by using the project property pages in the Visual Studio integrated development environment (IDE). For more information, see How to: Target a version of the .NET Framework. The available values for `TargetFrameworkVersion` are `v2.0`, `v3.0`, `v3.5`, `v4.5.2`, `v4.6`, `v4.6.1`, `v4.6.2`, `v4.7`, `v4.7.1`, `v4.7.2`, and `v4.8`.

```
<TargetFrameworkVersion>v4.0</TargetFrameworkVersion>
```

A *target profile* is a subset of a target framework. For example, the .NET Framework 4 Client profile does not include references to the MSBuild assemblies.

> **NOTE**
> Target profiles apply only to portable class libraries.

The target profile is specified in the `TargetFrameworkProfile` property in a project file. You can change the target profile by using the target-framework control in the project property pages in the IDE.

```
<TargetFrameworkVersion>v4.0</TargetFrameworkVersion>
<TargetFrameworkProfile>Client</TargetFrameworkProfile>
```

Target platform

A *platform* is combination of hardware and software that defines a particular runtime environment. For example,

- `x86` designates a 32-bit Windows operating system that is running on an Intel 80x86 processor or its equivalent.
- `x64` designates a 64-bit Windows operating system that is running on an Intel x64 processor or it equivalent.
- `Xbox` designates the Microsoft Xbox 360 platform.

A *target platform* is the particular platform that your project is built to run on. The target platform is specified in the `PlatformTarget` build property in a project file. You can change the target platform by using the project property pages or the **Configuration Manager** in the IDE.

```
<PropertyGroup>
   <PlatformTarget>x86</PlatformTarget>
</PropertyGroup>
```

A *target configuration* is a subset of a target platform. For example, the `x86``Debug` configuration does not include most code optimizations. The target configuration is specified in the `Configuration` build property in a project file. You can change the target configuration by using the project property pages or the **Configuration Manager**.

```
<PropertyGroup>
   <PlatformTarget>x86</PlatformTarget>
   <Configuration>Debug</Configuration>
<PropertyGroup>
```

See also

- Multitargeting

Resolve assemblies at design time

9/24/2019 • 2 minutes to read • Edit Online

When you add a reference to an assembly through the **.NET** tab of the **Add Reference** dialog, the reference points to an intermediate reference assembly; that is, an assembly that contains all the type and signature information, but that doesn't necessarily contain any code. The **.NET** tab lists reference assemblies that correspond to runtime assemblies in the .NET Framework. In addition, it lists reference assemblies that correspond to runtime assemblies in the registered AssemblyFoldersEx folders that are used by third parties.

Multi-targeting

Visual Studio 2013 lets you target versions of the .NET Framework that run either on the Common Language Runtime (CLR) version 2.0 or version 4. These versions include .NET Framework versions 2.0, 3.0, 3.5, 4, 4.5, and 4.5.1, and Silverlight versions 1.0, 2.0, and 3.0. If a new .NET Framework version that is based on CLR version 2.0 or version 4 is released, the Framework can be installed by using a targeting pack, and it will automatically show up as a target in Visual Studio.

How type resolution works

At run time, the CLR resolves the types in the assembly by looking in the GAC, the *bin* directory, and in any probing paths. This is handled by the fusion loader. But, how does the fusion loader know what it is looking for? It depends on a resolution made at design time, when the application is built.

During the build, the compiler resolves application types by using reference assemblies. In .NET Framework versions 2.0, 3.0, 3.5, 4, 4.5, and 4.5.1, the reference assemblies install when the .NET Framework installs.

The reference assemblies are supplied by the targeting pack that ships with the corresponding version of the .NET Framework SDK. The Framework itself provides only the runtime assemblies. In order to build applications, you need to install both the .NET Framework and the corresponding .NET Framework SDK.

When you target a specific .NET Framework, the build system resolves all types by using the reference assemblies in the targeting pack. At run time, the fusion loader resolves these same types to the runtime assemblies, which are typically located in the GAC.

If reference assemblies are not available, then the build system resolves assembly types by using the runtime assemblies. Because runtime assemblies in the GAC aren't distinguished by minor version numbers, it's possible that resolution will be made to the wrong assembly. This could happen, for example, if a new method introduced in the .NET Framework version 3.5 is referenced while targeting version 3.0. The build will succeed, and the application will run on the build machine, but will fail when deployed to a machine that does not have version 3.5 installed.

The targeting pack that now ships with the .NET Framework SDK includes a list of all of the runtime assemblies in that version of the Framework, called the redistribution (redist) list, making it impossible for the build system to resolve types against the wrong version of the assembly.

See also

- Advanced concepts

Configure targets and tasks

10/24/2019 • 2 minutes to read • Edit Online

You can configure MSBuild targets and tasks to run out-of-process with MSBuild so that you can target contexts that differ from the one you are running on. For example, you can target a 32-bit .NET Framework 2.0 application while the development computer is running on a 64-bit .NET Framework 4.5 operating system. You can also target computers that run with the .NET Framework 4 or earlier. The combination of 32- or 64-bitness and the specific .NET Framework version is known as the *target context*.

Installation

The .NET Framework 4.5 and 4.5.1 replace the common language runtime (CLR), targets, tasks, and tools of the .NET Framework 4 without renaming them. The .NET Framework 4.5.1 is installed as part of Visual Studio 2013.

If you want to install MSBuild separately from Visual Studio, you can download the installation package from MSBuild download. You must also install the .NET Framework versions you wish to use.

Targets and tasks

MSBuild runs certain build tasks out of process to target a larger set of contexts. For example, a 32-bit MSBuild might run a build task in a 64-bit process to target a 64-bit computer. This is controlled by `UsingTask` arguments and `Task` parameters. The targets installed by the .NET Framework 4.5 set these arguments and parameters, and no changes are required to build applications for the various target contexts.

If you want to create your own target context, you must set these arguments and parameters appropriately. Look in the .NET Framework 4.5 *Microsoft.Common.targets* file and the *Microsoft.Common.Tasks* file for examples. For information about how to create a custom task that can work with multiple target contexts, or how to modify existing tasks, see How to: Configure targets and tasks.

See also

- Multitargeting

How to: Configure targets and tasks

2/21/2019 • 3 minutes to read • Edit Online

Selected MSBuild tasks can be set to run in the environment they target, regardless of the environment of the development computer. For example, when you use a 64-bit computer to build an application that targets a 32-bit architecture, selected tasks are run in a 32-bit process.

> **NOTE**
> If a build task is written in a .NET language, such as Visual C# or Visual Basic, and does not use native resources or tools, then it will run in any target context without adaptation.

UsingTask attributes and task parameters

The following `UsingTask` attributes affect all operations of a task in a particular build process:

- The `Runtime` attribute, if present, sets the common language runtime (CLR) version, and can take any one of these values: `CLR2`, `CLR4`, `CurrentRuntime`, or `*` (any runtime).
- The `Architecture` attribute, if present, sets the platform and bitness, and can take any one of these values: `x86`, `x64`, `CurrentArchitecture`, or `*` (any architecture).
- The `TaskFactory` attribute, if present, sets the task factory that creates and runs the task instance, and takes only the value `TaskHostFactory`. For more information, see Task factories later in this document.

```
<UsingTask TaskName="SimpleTask"
    Runtime="CLR2"
    Architecture="x86"
    AssemblyFile="$(MSBuildToolsPath)\Microsoft.Build.Tasks.v3.5.dll" />
```

You can also use the `MSBuildRuntime` and `MSBuildArchitecture` parameters to set the target context of an individual task.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <Target Name="MyTarget">
        <SimpleTask MSBuildRuntime="CLR2" MSBuildArchitecture= "x86"/>
    </Target>
</Project>
```

Before MSBuild runs a task, it looks for a matching `UsingTask` that has the same target context. Parameters that are specified in the `UsingTask` but not in the corresponding task are considered to be matched. Parameters that specified in the task but not in the corresponding `UsingTask` are also considered to be matched. If parameter values are not specified in either the `UsingTask` or the task, the values default to `*` (any parameter).

> **WARNING**
> If more than one `UsingTask` exists and all have matching `TaskName`, `Runtime`, and `Architecture` attributes, the last one to be evaluated replaces the others.

If parameters are set on the task, MSBuild attempts to find a `UsingTask` that matches these parameters or, at least,

is not in conflict with them. More than one `UsingTask` can specify the target context of the same task. For example, a task that has different executables for different target environments might resemble this one:

```
<UsingTask TaskName="MyTool"
    Runtime="CLR2"
    Architecture="x86"
    AssemblyFile="$(MyToolsPath)\MyTool.v2.0.dll" />

<UsingTask TaskName="MyTool"
    Runtime="CLR4"
    Architecture="x86"
    AssemblyFile="$(MyToolsPath)\MyTool.4.0.dll" />

<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <Target Name="MyTarget">
        <MyTool MSBuildRuntime="CLR2" MSBuildArchitecture= "x86"/>
    </Target>
</Project>
```

Task factories

Before it runs a task, MSBuild checks to see whether it is designated to run in the current software context. If the task is so designated, MSBuild passes it to the AssemblyTaskFactory, which runs it in the current process; otherwise, MSBuild passes the task to the TaskHostFactory, which runs the task in a process that matches the target context. Even if the current context and the target context match, you can force a task to run out-of-process (for isolation, security, or other reasons) by setting `TaskFactory` to `TaskHostFactory`.

```
<UsingTask TaskName="MisbehavingTask"
    TaskFactory="TaskHostFactory"
    AssemblyFile="$(MSBuildToolsPath)\MyTasks.dll">
</UsingTask>
```

Phantom task parameters

Like any other task parameters, `MSBuildRuntime` and `MSBuildArchitecture` can be set from build properties.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <PropertyGroup>
        <FrameworkVersion>3.0</FrameworkVersion>
    </PropertyGroup>
    <Target Name="MyTarget">
        <SimpleTask MSBuildRuntime="$(FrameworkVerion)" MSBuildArchitecture= "x86"/>
    </Target>
</Project>
```

Unlike other task parameters, `MSBuildRuntime` and `MSBuildArchitecture` are not apparent to the task itself. To write a task that is aware of the context in which it runs, you must either test the context by calling the .NET Framework, or use build properties to pass the context information through other task parameters.

> **NOTE**
>
> `UsingTask` attributes can be set from toolset and environment properties.

The `MSBuildRuntime` and `MSBuildArchitecture` parameters provide the most flexible way to set the target context, but also the most limited in scope. On the one hand, because they are set on the task instance itself and are not

evaluated until the task is about to run, they can derive their value from the full scope of properties available at both evaluation-time and build-time. On the other hand, these parameters only apply to a particular instance of a task in a particular target.

> **NOTE**
>
> Task parameters are evaluated in the context of the parent node, not in the context of the task host. Environment variables that are runtime- or architecture- dependent (such as the *Program Files* location) will evaluate to the value that matches the parent node. However, if the same environment variable is read directly by the task, it will correctly be evaluated in the context of the task host.

See also

- Configure targets and tasks

Troubleshoot .NET Framework targeting errors

10/31/2019 • 2 minutes to read • Edit Online

This topic describes MSBuild errors that might occur because of reference issues and how you can resolve those errors.

You have referenced a project or assembly that targets a different version of the .NET Framework

You can create applications that reference projects or assemblies that target different versions of the .NET Framework. For example, you can create an application that targets the client profile for the .NET Framework 4 but references an assembly that targets the .NET Framework 2.0. However, if you create a project that targets an earlier version of the .NET Framework, you can't set a reference in that project to a project or assembly that targets the client profile for the .NET Framework 4 or the .NET Framework 4 itself. To resolve the error, make sure that your application targets a profile or profiles that are compatible with the profile that's targeted by the projects or assemblies that your application references.

You have re-targeted a project to a different version of the .NET Framework

If you change the target version of the .NET Framework for your application, Visual Studio changes some of the references, but you may have to update some references manually. For example, one of the previously mentioned errors might occur if you change an application to target the .NET Framework 3.5 Service Pack 1 and that application has resources or settings that rely on the client profile for the .NET Framework 4.

To work around application settings, open **Solution Explorer**, choose **Show All Files**, and then edit the *app.config* file in the XML editor of Visual Studio. Change the version in the settings to match the appropriate version of the .NET Framework. For example, you can change the version setting from 4.0.0.0 to 2.0.0.0. Similarly, for an application that has added resources, open **Solution Explorer**, choose the **Show All Files** button, expand **My Project** (Visual Basic) or **Properties** (C#), and then edit the *Resources.resx* file in the XML editor of Visual Studio. Change the version setting from 4.0.0.0 to 2.0.0.0.

If your application has resources such as icons or bitmaps or settings such as data connection strings, you can also resolve the error by removing all the items on the **Settings** page of the **Project Designer** and then re-adding the required settings.

You have re-targeted a project to a different version of the .NET Framework and references do not resolve

If you retarget a project to a different version of the .NET Framework, your references may not resolve properly in some cases. Explicit fully qualified references to assemblies often cause this issue, but you can resolve it by removing the references that do not resolve and then adding them back to the project. As an alternative, you can edit the project file to replace the references. First, you remove references of the following form:

```
<Reference Include="System.ServiceModel, Version=3.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089,
processorArchitecture=MSIL" />
```

Then you replace them with the simple form:

```
<Reference Include="System.ServiceModel" />
```

> **NOTE**
> After you close and reopen your project, you should also rebuild it to ensure that all references resolve correctly.

See also

- How to: Target a version of the .NET Framework
- .NET Framework client profile
- Framework targeting overview
- Multitargeting

File tracking

2/21/2019 • 2 minutes to read • Edit Online

File tracking logs calls to the Windows file system for a process and its child processes. By calling the functions listed below, programs control when to turn this logging on and off and specify the log file to use.

- EndTrackingContext Stop tracking the current context.
- ResumeTracking Resume tracking after a call to SuspendTracking.
- SetThreadCount Set the number of threads to use for tracking.
- StartTrackingContext Begin a new tracking context.
- StartTrackingContextWithRoot Begin a new tracking context with a specified root.
- StopTrackingAndCleanup End tracking and release resources used.
- SuspendTracking Temporarily suspend tracking.
- WriteAllTLogs Write out the tracking logs for all contexts.
- WriteContextTLogs Write out the tracking log for the current context.

EndTrackingContext

2/21/2019 • 2 minutes to read • Edit Online

End the current tracking context.

Syntax

```
HRESULT WINAPI EndTrackingContext();
```

Return value

An **HRESULT** with the **SUCCEEDED** bit set if the tracking context was ended.

Requirements

Header: *FileTracker.h*

See also

- StartTrackingContext

ResumeTracking

2/21/2019 • 2 minutes to read • Edit Online

Resumes tracking in the current context.

Syntax

```
HRESULT WINAPI ResumeTracking();
```

Return value

An **HRESULT** with the **SUCCEEDED** bit set if tracking was resumed. **E_FAIL** is returned if tracking cannot be resumed because the context was not available.

Requirements

Header: *FileTracker.h*

See also

- SuspendTracking

SetThreadCount

2/22/2019 • 2 minutes to read • Edit Online

Sets the global thread count, and assigns that count to the current thread.

Syntax

```
HRESULT WINAPI SetThreadCount(int threadCount);
```

Parameters

[in] `threadCount`

The number of threads to use.

Return value

An **HRESULT** with the **SUCCEEDED** bit set if the thread count was updated.

Requirements

Header: *FileTracker.h*

StartTrackingContext

2/22/2019 • 2 minutes to read • Edit Online

Start a tracking context.

Syntax

```
HRESULT WINAPI StartTrackingContext(LPCTSTR intermediateDirectory, LPCTSTR taskName);
```

Parameters

[in] `intermediateDirectory`

The directory in which to store the tracking log.

[in] `taskName`

Identifies the tracking context. This name is used to create the log file name.

Return value

An **HRESULT** with the **SUCCEEDED** bit set if the tracking context was created.

Requirements

Header: *FileTracker.h*

StartTrackingContextWithRoot

2/22/2019 • 2 minutes to read • Edit Online

Starts a tracking context using a response file specifying a root marker.

Syntax

```
HRESULT WINAPI StartTrackingContextWithRoot(LPCTSTR intermediateDirectory, LPCTSTR taskName, LPCTSTR
rootMarkerResponseFile);
```

Parameters

[in] `intermediateDirectory`

The directory in which to store the tracking log.

[in] `taskName`

Identifies the tracking context. This name is used to create the log file name.

[in] `rootMarkerResponseFile`

The pathname of a response file containing a root marker. The root name is used to group all tracking for a context together.

Return value

An **HRESULT** with the **SUCCEEDED** bit set if the tracking context was created.

Requirements

Header: *FileTracker.h*

See also

- StartTrackingContext

StopTrackingAndCleanup

2/21/2019 • 2 minutes to read • Edit Online

Stops all tracking and frees any memory used by the tracking session.

Syntax

```
HRESULT WINAPI StopTrackingAndCleanup(void);
```

Return value

Returns an **HRESULT** with the **SUCCEEDED** bit set if tracking was stopped.

Requirements

Header: *FileTracker.h*

See also

- StartTrackingContext

SuspendTracking

2/21/2019 • 2 minutes to read • Edit Online

Suspends tracking in the current context.

Syntax

```
HRESULT WINAPI SuspendTracking(void);
```

Return value

An **HRESULT** with the **SUCCEEDED** bit set if tracking was suspended.

Requirements

Header: *FileTracker.h*

See also

- ResumeTracking

WriteAllTLogs

2/22/2019 • 2 minutes to read • Edit Online

Writes tracking logs for all threads and contexts.

Syntax

```
HRESULT WINAPI WriteAllTLogs(LPCTSTR intermediateDirectory, LPCTSTR tlogRootName);
```

Parameters

[in] `intermediateDirectory`

The directory in which to store the tracking log.

[in] `tlogRootName`

The root name of the log file name.

Return value

An **HRESULT** with the **SUCCEEDED** bit set if the tracking context was created.

Requirements

Header: *FileTracker.h*

See also

- WriteContextTLogs

WriteContextTLogs

2/22/2019 • 2 minutes to read • Edit Online

Writes logs files for the current context.

Syntax

```
HRESULT WINAPI WriteContextTLogs(LPCTSTR intermediateDirectory, LPCTSTR tlogRootName);
```

Parameters

[in] `intermediateDirectory`

The directory in which to store the tracking log.

[in] `tlogRootName`

The root name of the log file name.

Return value

An **HRESULT** with the **SUCCEEDED** bit set if the tracking context was created.

Requirements

Header: *FileTracker.h*

See also

- WriteAllTLogs

Customize your build

6/14/2019 • 6 minutes to read • Edit Online

MSBuild projects that use the standard build process (importing *Microsoft.Common.props* and *Microsoft.Common.targets*) have several extensibility hooks that you can use to customize your build process.

Add arguments to command-line MSBuild invocations for your project

A *Directory.Build.rsp* file in or above your source directory will be applied to command-line builds of your project. For details, see MSBuild response files.

Directory.Build.props and Directory.Build.targets

Prior to MSBuild version 15, if you wanted to provide a new, custom property to projects in your solution, you had to manually add a reference to that property to every project file in the solution. Or, you had to define the property in a *.props* file and then explicitly import the *.props* file in every project in the solution, among other things.

However, now you can add a new property to every project in one step by defining it in a single file called *Directory.Build.props* in the root folder that contains your source. When MSBuild runs, *Microsoft.Common.props* searches your directory structure for the *Directory.Build.props* file (and *Microsoft.Common.targets* looks for *Directory.Build.targets*). If it finds one, it imports the property. *Directory.Build.props* is a user-defined file that provides customizations to projects under a directory.

> **NOTE**
> Linux-based file systems are case-sensitive. Make sure the casing of the Directory.Build.props filename matches exactly, or it won't be detected during the build process.
>
> See this GitHub issue for more information.

Directory.Build.props example

For example, if you wanted to enable all of your projects to access the new Roslyn **/deterministic** feature (which is exposed in the Roslyn `CoreCompile` target by the property `$(Deterministic)`), you could do the following.

1. Create a new file in the root of your repo called *Directory.Build.props*.
2. Add the following XML to the file.

```
<Project>
 <PropertyGroup>
   <Deterministic>true</Deterministic>
 </PropertyGroup>
</Project>
```

3. Run MSBuild. Your project's existing imports of *Microsoft.Common.props* and *Microsoft.Common.targets* find the file and import it.

Search scope

When searching for a *Directory.Build.props* file, MSBuild walks the directory structure upwards from your project location (`$(MSBuildProjectFullPath)`), stopping after it locates a *Directory.Build.props* file. For example, if your `$(MSBuildProjectFullPath)` was *c:\users\username\code\test\case1*, MSBuild would start searching there and then

search the directory structure upward until it located a *Directory.Build.props* file, as in the following directory structure.

```
c:\users\username\code\test\case1
c:\users\username\code\test
c:\users\username\code
c:\users\username
c:\users
c:\
```

The location of the solution file is irrelevant to *Directory.Build.props*.

Import order

Directory.Build.props is imported very early in *Microsoft.Common.props*, and properties defined later are unavailable to it. So, avoid referring to properties that are not yet defined (and will evaluate to empty).

Directory.Build.targets is imported from *Microsoft.Common.targets* after importing *.targets* files from NuGet packages. So, it can override properties and targets defined in most of the build logic, but sometimes you may need to customize the project file after the final import.

Use case: multi-level merging

Suppose you have this standard solution structure:

```
\
  MySolution.sln
  Directory.Build.props     (1)
  \src
    Directory.Build.props   (2-src)
    \Project1
    \Project2
  \test
    Directory.Build.props   (2-test)
    \Project1Tests
    \Project2Tests
```

It might be desirable to have common properties for all projects *(1)*, common properties for *src* projects *(2-src)*, and common properties for *test* projects *(2-test)*.

To make MSBuild correctly merge the "inner" files (*2-src* and *2-test*) with the "outer" file (*1*), you must take into account that once MSBuild finds a *Directory.Build.props* file, it stops further scanning. To continue scanning and merge into the outer file, place this code into both inner files:

```
<Import Project="$([MSBuild]::GetPathOfFileAbove('Directory.Build.props', '$(MSBuildThisFileDirectory)../'))"
/>
```

A summary of MSBuild's general approach is as follows:

- For any given project, MSBuild finds the first *Directory.Build.props* upward in the solution structure, merges it with defaults, and stops scanning for more
- If you want multiple levels to be found and merged, then `<Import...>` (shown above) the "outer" file from the "inner" file
- If the "outer" file does not itself also import something above it, then scanning stops there
- To control the scanning/merging process, use `$(DirectoryBuildPropsPath)` and `$(ImportDirectoryBuildProps)`

Or more simply: the first *Directory.Build.props* that doesn't import anything is where MSBuild stops.

Choose between adding properties to a .props or .targets file

MSBuild is import-order dependent, and the last definition of a property (or a `UsingTask` or target) is the

definition used.

When using explicit imports, you can import from a *.props* or *.targets* file at any point. Here is the widely used convention:

- *.props* files are imported early in the import order.
- *.targets* files are imported late in the build order.

This convention is enforced by `<Project Sdk="SdkName">` imports (that is, the import of *Sdk.props* comes first, before all of the contents of the file, then *Sdk.targets* comes last, after all of the contents of the file).

When deciding where to put the properties, use the following general guidelines:

- For many properties, it doesn't matter where they're defined, because they're not overwritten and will be read only at execution time.
- For behavior that might be customized in an individual project, set defaults in *.props* files.
- Avoid setting dependent properties in *.props* files by reading the value of a possibly customized property, because the customization won't happen until MSBuild reads the user's project.
- Set dependent properties in *.targets* files, because they'll pick up customizations from individual projects.
- If you need to override properties, do it in a *.targets* file, after all user-project customizations have had a chance to take effect. Be cautious when using derived properties; derived properties may need to be overridden as well.
- Include items in *.props* files (conditioned on a property). All properties are considered before any item, so user-project property customizations get picked up, and this gives the user's project the opportunity to `Remove` or `Update` any item brought in by the import.
- Define targets in *.targets* files. However, if the *.targets* file is imported by an SDK, remember that this scenario makes overriding the target more difficult because the user's project doesn't have a place to override it by default.
- If possible, prefer customizing properties at evaluation time over changing properties inside a target. This guideline makes it easier to load a project and understand what it's doing.

MSBuildProjectExtensionsPath

By default, *Microsoft.Common.props* imports `$(MSBuildProjectExtensionsPath)$(MSBuildProjectFile).*.props` and *Microsoft.Common.targets* imports `$(MSBuildProjectExtensionsPath)$(MSBuildProjectFile).*.targets`. The default value of `MSBuildProjectExtensionsPath` is `$(BaseIntermediateOutputPath)`, `obj/`. NuGet uses this mechanism to refer to build logic delivered with packages; that is, at restore time, it creates `{project}.nuget.g.props` files that refer to the package contents.

You can disable this extensibility mechanism by setting the property `ImportProjectExtensionProps` to `false` in a *Directory.Build.props* or before importing *Microsoft.Common.props*.

> **NOTE**
> Disabling MSBuildProjectExtensionsPath imports will prevent build logic delivered in NuGet packages from applying to your project. Some NuGet packages require build logic to perform their function and will be rendered useless when this is disabled.

.user file

Microsoft.Common.CurrentVersion.targets imports `$(MSBuildProjectFullPath).user` if it exists, so you can create a

file next to your project with that additional extension. For long-term changes you plan to check into source control, prefer changing the project itself, so that future maintainers do not have to know about this extension mechanism.

MSBuildExtensionsPath and MSBuildUserExtensionsPath

> **WARNING**
> Using these extension mechanisms makes it harder to get repeatable builds across machines. Try to use a configuration that can be checked into your source control system and shared among all developers of your codebase.

By convention, many core build logic files import

```
$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\{TargetFileName}\ImportBefore\*.targets
```

before their contents, and

```
$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\{TargetFileName}\ImportAfter\*.targets
```

afterward. This convention allows installed SDKs to augment the build logic of common project types.

The same directory structure is searched in `$(MSBuildUserExtensionsPath)`, which is the per-user folder *%LOCALAPPDATA%\Microsoft\MSBuild*. Files placed in that folder will be imported for all builds of the corresponding project type run under that user's credentials. You can disable the user extensions by setting properties named after the importing file in the pattern `ImportUserLocationsByWildcardBefore{ImportingFileNameWithNoDots}`. For example, setting `ImportUserLocationsByWildcardBeforeMicrosoftCommonProps` to `false` would prevent importing `$(MSBuildUserExtensionsPath)\$(MSBuildToolsVersion)\Imports\Microsoft.Common.props\ImportBefore\*`.

Customize the solution build

> **IMPORTANT**
> Customizing the solution build in this way applies only to command-line builds with *MSBuild.exe*. It **does not** apply to builds inside Visual Studio.

When MSBuild builds a solution file, it first translates it internally into a project file and then builds that. The generated project file imports `before.{solutionname}.sln.targets` before defining any targets and `after.{solutionname}.sln.targets` after importing targets, including targets installed to the `$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\SolutionFile\ImportBefore` and `$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\SolutionFile\ImportAfter` directories.

For example, you could define a new target to write a custom log message after building *MyCustomizedSolution.sln* by creating a file in the same directory named *after.MyCustomizedSolution.sln.targets* that contains

```
<Project>
 <Target Name="EmitCustomMessage" AfterTargets="Build">
   <Message Importance="High" Text="The solution has completed the Build target" />
 </Target>
</Project>
```

See also

- MSBuild concepts
- MSBuild reference

MSBuild best practices

4/18/2019 • 2 minutes to read • Edit Online

We recommend the following best practices for writing MSBuild scripts:

- Default property values are best handled by using the `Condition` attribute, and not by declaring a property whose default value can be overridden on the command line. For example, use

```
<MyProperty Condition="'$(MyProperty)' == ''">
   MyDefaultValue
</MyProperty>
```

- Avoid wildcards when you select items. Instead, specify files explicitly. This makes it easier to track down errors that may occur when you add or delete files.

See also

- Advanced concepts

Logging in MSBuild

2/21/2019 • 2 minutes to read • Edit Online

Logging provides a way for you to monitor the progress of a build. Logging captures build events, messages, warnings, and errors in a log file.

In this section

- Obtain build logs

Describes the various aspects of logging in MSBuild.

- Build loggers

Outlines the steps required to create a single-processor logger.

- Logging in a multi-processor environment

Describes how logging works in a multi-processor environment and the two multi-processor logging models.

- Write multi-processor-aware loggers

Outlines how to create multi-processor-aware loggers and how to use the ConfigurableForwardingLogger.

- Create forwarding loggers

Outlines how to create custom forwarding loggers.

See also

- Build multiple projects in parallel Describes how to build multiple projects faster by running them in parallel.

Obtain build logs with MSBuild

4/8/2019 • 2 minutes to read • Edit Online

By using switches with MSBuild, you can specify how much build data you want to review and whether you want to save build data to one or more files. You can also specify a custom logger to collect build data. For information about MSBuild command-line switches that this topic doesn't cover, see Command-line reference.

> **NOTE**
> If you build projects by using the Visual Studio IDE, you can troubleshoot those builds by reviewing build logs. For more information, see How to: View, save, and configure build log files.

Set the level of detail

When you build a project by using MSBuild without specifying a level of detail, the following information appears in the output log:

- Errors, warnings, and messages that are categorized as highly important.
- Some status events.
- A summary of the build.

By using the **-verbosity** (**-v**) switch, you can control how much data appears in the output log. For troubleshooting, use a verbosity level of either `detailed` (`d`) or `diagnostic` (`diag`), which provides the most information.

The build process may be slower when you set the **-verbosity** to `detailed` and even slower when you set the **-verbosity** to `diagnostic`.

```
msbuild MyProject.proj -t:go -v:diag
```

Verbosity settings

The following table shows how the log verbosity (column values) affects which types of message (row values) are logged.

	QUIET	MINIMAL	NORMAL	DETAILED	DIAGNOSTIC
Errors	□	□	□	□	□
Warnings	□	□	□	□	□
High-importance Messages		□	□	□	□
Normal-importance Messages			□	□	□

	QUIET	MINIMAL	NORMAL	DETAILED	DIAGNOSTIC
Low-importance Messages				□	□
Additional MSBuild-engine information					□

Save the build log to a file

You can use the **-fileLogger** (**fl**) switch to save build data to a file. The following example saves build data to a file that's named *msbuild.log*.

```
msbuild MyProject.proj -t:go -fileLogger
```

In the following example, the log file is named *MyProjectOutput.log*, and the verbosity of the log output is set to `diagnostic`. You specify those two settings by using the **-filelogparameters** (`flp`) switch.

```
msbuild MyProject.proj -t:go -fl -flp:logfile=MyProjectOutput.log;verbosity=diagnostic
```

For more information, see Command-line reference.

Save the log output to multiple files

The following example saves the entire log to *msbuild1.log*, just the errors to *JustErrors.log*, and just the warnings to *JustWarnings.log*. The example uses file numbers for each of the three files. The file numbers are specified just after the **-fl** and **-flp** switches (for example, `-fl1` and `-flp1`).

The **-filelogparameters** (`flp`) switches for files 2 and 3 specify what to name each file and what to include in each file. No name is specified for file 1, so the default name of *msbuild1.log* is used.

```
msbuild MyProject.proj -t:go -fl1 -fl2 -fl3 -flp2:logfile=JustErrors.log;errorsonly -
flp3:logfile=JustWarnings.log;warningsonly
```

For more information, see Command-line reference.

Save a binary log

You can save the log in compressed, binary format using the **-binaryLogger** (**bl**) switch. This log includes a detailed description of the build process and can be read by certain log analysis tools.

In the following example, a binary log file is created with the name *binarylogfilename*.

```
-bl:binarylogfilename.binlog
```

For more information, see Command-line reference.

Use a custom logger

You can write your own logger by authoring a managed type that implements the ILogger interface. You might use a custom logger, for instance, to send build errors in email, log them to a database, or log them to an XML

file. For more information, see Build loggers.

In the MSBuild command line, you specify the custom logger by using the **-logger** switch. You can also use the **-noconsolelogger** switch to disable the default console logger.

See also

- LoggerVerbosity
- Build loggers
- Logging in a multi-processor environment
- Creating forwarding loggers
- MSBuild concepts

Build loggers

2/21/2019 • 5 minutes to read • Edit Online

Loggers provide a way for you to customize the output of your build and display messages, errors, or warnings in response to specific build events. Each logger is implemented as a .NET class that implements the ILogger interface, which is defined in the *Microsoft.Build.Framework.dll* assembly.

There are two approaches you can use when implementing a logger:

- Implement the ILogger interface directly.
- Derive your class from the helper class, Logger, which is defined in the *Microsoft.Build.Utilities.dll* assembly. Logger implements ILogger and provides default implementations of some ILogger members.

  This topic will explain how to write a simple logger that derives from Logger, and displays messages on the console in response to certain build events.

Register for events

The purpose of a logger is to gather information on build progress as it is reported by the build engine, and then report that information in a useful way. All loggers must override the Initialize method, which is where the logger registers for events. In this example, the logger registers for the TargetStarted, ProjectStarted, and ProjectFinished events.

```
public class MySimpleLogger : Logger
{
    public override void Initialize(Microsoft.Build.Framework.IEventSource eventSource)
    {
        //Register for the ProjectStarted, TargetStarted, and ProjectFinished events
        eventSource.ProjectStarted += new ProjectStartedEventHandler(eventSource_ProjectStarted);
        eventSource.TargetStarted += new TargetStartedEventHandler(eventSource_TargetStarted);
        eventSource.ProjectFinished += new ProjectFinishedEventHandler(eventSource_ProjectFinished);
    }
 }
```

Respond to events

Now that the logger is registered for specific events, it needs to handle those events when they occur. For the ProjectStarted, and ProjectFinished events, the logger simply writes a short phrase and the name of the project file involved in the event. All messages from the logger are written to the console window.

```
void eventSource_ProjectStarted(object sender, ProjectStartedEventArgs e)
{
    Console.WriteLine("Project Started: " + e.ProjectFile);
}

void eventSource_ProjectFinished(object sender, ProjectFinishedEventArgs e)
{
    Console.WriteLine("Project Finished: " + e.ProjectFile);
}
```

Respond to logger verbosity values

In some cases, you may want to only log information from an event if the MSBuild.exe **-verbosity** switch contains a certain value. In this example, the TargetStarted event handler only logs a message if the Verbosity property, which is set by the **-verbosity** switch, is equal to LoggerVerbosity `Detailed`.

```
void eventSource_TargetStarted(object sender, TargetStartedEventArgs e)
{
    if (Verbosity == LoggerVerbosity.Detailed)
    {
        Console.WriteLine("Target Started: " + e.TargetName);
    }
}
```

Specify a logger

Once the logger is compiled into an assembly, you need to tell MSBuild to use that logger during builds. This is done using the **-logger** switch with *MSBuild.exe*. For more information on the switches available for *MSBuild.exe*, see Command-line reference.

The following command line builds the project *MyProject.csproj* and uses the logger class implemented in *SimpleLogger.dll*. The **-nologo** switch hides the banner and copyright message and the **-noconsolelogger** switch disables the default MSBuild console logger.

```
MSBuild -nologo -noconsolelogger -logger:SimpleLogger.dll
```

The following command line builds the project with the same logger, but with a `Verbosity` level of `Detailed`.

```
MSBuild -nologo -noconsolelogger -logger:SimpleLogger.dll -verbosity:Detailed
```

Example

Description

The following example contains the complete code for the logger.

Code

```
using System;
using Microsoft.Build.Utilities;
using Microsoft.Build.Framework;

namespace SimpleLogger
{

    public class MySimpleLogger : Logger
    {
        public override void Initialize(Microsoft.Build.Framework.IEventSource eventSource)
        {
            //Register for the ProjectStarted, TargetStarted, and ProjectFinished events
            eventSource.ProjectStarted += new ProjectStartedEventHandler(eventSource_ProjectStarted);
            eventSource.TargetStarted += new TargetStartedEventHandler(eventSource_TargetStarted);
            eventSource.ProjectFinished += new ProjectFinishedEventHandler(eventSource_ProjectFinished);
        }

        void eventSource_ProjectStarted(object sender, ProjectStartedEventArgs e)
        {
            Console.WriteLine("Project Started: " + e.ProjectFile);
        }

        void eventSource_ProjectFinished(object sender, ProjectFinishedEventArgs e)
        {
            Console.WriteLine("Project Finished: " + e.ProjectFile);
        }
        void eventSource_TargetStarted(object sender, TargetStartedEventArgs e)
        {
            if (Verbosity == LoggerVerbosity.Detailed)
            {
                Console.WriteLine("Target Started: " + e.TargetName);
            }
        }
    }
}
```

Example

Description

The following example shows how to implement a logger that writes the log to a file rather than displaying it in the console window.

Code

```
using System;
using System.IO;
using System.Security;
using Microsoft.Build.Framework;
using Microsoft.Build.Utilities;

namespace MyLoggers
{
    // This logger will derive from the Microsoft.Build.Utilities.Logger class,
    // which provides it with getters and setters for Verbosity and Parameters,
    // and a default empty Shutdown() implementation.
    public class BasicFileLogger : Logger
    {
        /// <summary>
        /// Initialize is guaranteed to be called by MSBuild at the start of the build
        /// before any events are raised.
        /// </summary>
        public override void Initialize(IEventSource eventSource)
        {
            // The name of the log file should be passed as the first item in the
```

```
            // "parameters" specification in the /logger switch.  It is required
            // to pass a log file to this logger. Other loggers may have zero or more than
            // one parameters.
            if (null == Parameters)
            {
                throw new LoggerException("Log file was not set.");
            }
            string[] parameters = Parameters.Split(';');

            string logFile = parameters[0];
            if (String.IsNullOrEmpty(logFile))
            {
                throw new LoggerException("Log file was not set.");
            }

            if (parameters.Length > 1)
            {
                throw new LoggerException("Too many parameters passed.");
            }

            try
            {
                // Open the file
                this.streamWriter = new StreamWriter(logFile);
            }
            catch (Exception ex)
            {
                if
                (
                    ex is UnauthorizedAccessException
                    || ex is ArgumentNullException
                    || ex is PathTooLongException
                    || ex is DirectoryNotFoundException
                    || ex is NotSupportedException
                    || ex is ArgumentException
                    || ex is SecurityException
                    || ex is IOException
                )
                {
                    throw new LoggerException("Failed to create log file: " + ex.Message);
                }
                else
                {
                    // Unexpected failure
                    throw;
                }
            }

            // For brevity, we'll only register for certain event types. Loggers can also
            // register to handle TargetStarted/Finished and other events.
            eventSource.ProjectStarted += new ProjectStartedEventHandler(eventSource_ProjectStarted);
            eventSource.TaskStarted += new TaskStartedEventHandler(eventSource_TaskStarted);
            eventSource.MessageRaised += new BuildMessageEventHandler(eventSource_MessageRaised);
            eventSource.WarningRaised += new BuildWarningEventHandler(eventSource_WarningRaised);
            eventSource.ErrorRaised += new BuildErrorEventHandler(eventSource_ErrorRaised);
            eventSource.ProjectFinished += new ProjectFinishedEventHandler(eventSource_ProjectFinished);
        }

        void eventSource_ErrorRaised(object sender, BuildErrorEventArgs e)
        {
            // BuildErrorEventArgs adds LineNumber, ColumnNumber, File, amongst other parameters
            string line = String.Format(": ERROR {0}({1},{2}): ", e.File, e.LineNumber, e.ColumnNumber);
            WriteLineWithSenderAndMessage(line, e);
        }

        void eventSource_WarningRaised(object sender, BuildWarningEventArgs e)
        {
            // BuildWarningEventArgs adds LineNumber, ColumnNumber, File, amongst other parameters
            string line = String.Format(": Warning {0}({1},{2}): ", e.File, e.LineNumber, e.ColumnNumber);
```

```
            WriteLineWithSenderAndMessage(line, e);
        }

        void eventSource_MessageRaised(object sender, BuildMessageEventArgs e)
        {
            // BuildMessageEventArgs adds Importance to BuildEventArgs
            // Let's take account of the verbosity setting we've been passed in deciding whether to log the
message
            if ((e.Importance == MessageImportance.High && IsVerbosityAtLeast(LoggerVerbosity.Minimal))
                || (e.Importance == MessageImportance.Normal && IsVerbosityAtLeast(LoggerVerbosity.Normal))
                || (e.Importance == MessageImportance.Low && IsVerbosityAtLeast(LoggerVerbosity.Detailed))
                )
            {
                WriteLineWithSenderAndMessage(String.Empty, e);
            }
        }

        void eventSource_TaskStarted(object sender, TaskStartedEventArgs e)
        {
            // TaskStartedEventArgs adds ProjectFile, TaskFile, TaskName
            // To keep this log clean, this logger will ignore these events.
        }

        void eventSource_ProjectStarted(object sender, ProjectStartedEventArgs e)
        {
            // ProjectStartedEventArgs adds ProjectFile, TargetNames
            // Just the regular message string is good enough here, so just display that.
            WriteLine(String.Empty, e);
            indent++;
        }

        void eventSource_ProjectFinished(object sender, ProjectFinishedEventArgs e)
        {
            // The regular message string is good enough here too.
            indent--;
            WriteLine(String.Empty, e);
        }

        /// <summary>
        /// Write a line to the log, adding the SenderName and Message
        /// (these parameters are on all MSBuild event argument objects)
        /// </summary>
        private void WriteLineWithSenderAndMessage(string line, BuildEventArgs e)
        {
            if (0 == String.Compare(e.SenderName, "MSBuild", true /*ignore case*/))
            {
                // Well, if the sender name is MSBuild, let's leave it out for prettiness
                WriteLine(line, e);
            }
            else
            {
                WriteLine(e.SenderName + ": " + line, e);
            }
        }

        /// <summary>
        /// Just write a line to the log
        /// </summary>
        private void WriteLine(string line, BuildEventArgs e)
        {
            for (int i = indent; i > 0; i--)
            {
                streamWriter.Write("\t");
            }
            streamWriter.WriteLine(line + e.Message);
        }

        /// <summary>
        /// Shutdown() is guaranteed to be called by MSBuild at the end of the build, after all
```

```
        /// events have been raised.
        /// </summary>
        public override void Shutdown()
        {
            // Done logging, let go of the file
            streamWriter.Close();
        }

        private StreamWriter streamWriter;
        private int indent;
    }
}
```

See also

- Obtain build logs
- MSBuild concepts

Logging in a multi-processor environment

2/21/2019 • 3 minutes to read • Edit Online

The ability of MSBuild to use multiple processors can greatly decrease project building time, but it also adds complexity to logging. In a single-processor environment, the logger can handle incoming events, messages, warnings, and errors in a predictable, sequential manner. However, in a multi-processor environment, events from several sources can arrive simultaneously or out of sequence. MSBuild provides a new multi-processor-aware logger and enables the creation of custom "forwarding loggers."

Log multiple-processor builds

When you build one or more projects in a multi-processor or multi-core system, MSBuild build events for all the projects are generated simultaneously. An avalanche of event data may arrive at the logger at the same time or out of sequence. This can overwhelm the logger and cause increased build times, incorrect logger output, or even a broken build. To address these issues, the MSBuild logger can process out-of-sequence events and correlate events and their sources.

You can improve logging efficiency even more by creating a custom forwarding logger. A custom-forwarding logger acts as a filter by letting you choose, before you build, the events you want to monitor. When you use a custom forwarding logger, unwanted events do not overwhelm the logger, clutter your logs, or slow build times.

Central logging model

For multi-processor builds, MSBuild uses a "central logging model." In the central logging model, an instance of *MSBuild.exe* acts as the primary build process, or "central node." Secondary instances of *MSBuild.exe*, or "secondary nodes," are attached to the central node. Any ILogger-based loggers attached to the central node are known as "central loggers" and loggers attached to secondary nodes are known as "secondary loggers."

When a build occurs, the secondary loggers route their event traffic to the central loggers. Because events originate at several secondary nodes, the data arrives at the central node simultaneously but interleaved. To resolve event-to-project and event-to-target references, the event arguments include additional build event context information.

Although only ILogger is required to be implemented by the central logger, we recommend that you also implement INodeLogger if you want the central logger to initialize with the number of nodes that are participating in the build. The following overload of the Initialize method is invoked when the engine initializes the logger:

```
public interface INodeLogger: ILogger
{
    public void Initialize(IEventSource eventSource, int nodeCount);
}
```

Distributed logging model

In the central logging model, too much incoming message traffic, such as when many projects build at once, can overwhelm the central node, which stresses the system and lowers build performance.

To reduce this problem, MSBuild also enables a "distributed logging model" that extends the central logging model by letting you create forwarding loggers. A forwarding logger is attached to a secondary node and receives incoming build events from that node. The forwarding logger is just like a regular logger except that it can filter the events and then forward only the desired ones to the central node. This reduces the message traffic at the central node and therefore enables better performance.

You can create a forwarding logger by implementing the IForwardingLogger interface, which derives from ILogger. The interface is defined as:

```
public interface IForwardingLogger: INodeLogger
{
    public IEventRedirector EventRedirector { get; set; }
    public int NodeId { get; set; }
}
```

To forward events in a forwarding logger, call the ForwardEvent method of the IEventRedirector interface. Pass the appropriate BuildEventArgs, or a derivative, as the parameter.

For more information, see Create forwarding loggers.

Attaching a distributed logger

To attaching a distributed logger on a command line build, use the `-distributedlogger` (or, `-dl` for short) switch. The format for specifying the names of the logger types and classes are the same as those for the `-logger` switch, except that a distributed logger is comprised of two logging classes: a forwarding logger and a central logger. Following is an example of attaching a distributed logger:

```
msbuild.exe *.proj -distributedlogger:XMLCentralLogger,MyLogger,Version=1.0.2,
Culture=neutral*XMLForwardingLogger,MyLogger,Version=1.0.2,
Culture=neutral
```

An asterisk (*) separates the two logger names in the `-dl` switch.

See also

- Build loggers
- Create forwarding loggers

Write multi-processor-aware loggers

4/23/2019 • 4 minutes to read • Edit Online

The ability of MSBuild to take advantage of multiple processors can decrease project building time, but it also adds complexity to build event logging. In a single-processor environment, events, messages, warnings, and errors arrive at the logger in a predictable, sequential manner. However, in a multi-processor environment, events from different sources can arrive at the same time or out of sequence. To provide for this, MSBuild provides a multi-processor-aware logger and a new logging model, and lets you create custom "forwarding loggers."

Multi-processor logging challenges

When you build one or more projects on a multi-processor or multi-core system, MSBuild build events for all the projects are generated at the same time. An avalanche of event messages may arrive at the logger at the same time or out of sequence. Because a MSBuild 2.0 logger is not designed to handle this situation, it can overwhelm the logger and cause increased build times, incorrect logger output, or even a broken build. To address these issues, the logger (starting in MSBuild 3.5) can process out-of-sequence events and correlate events and their sources.

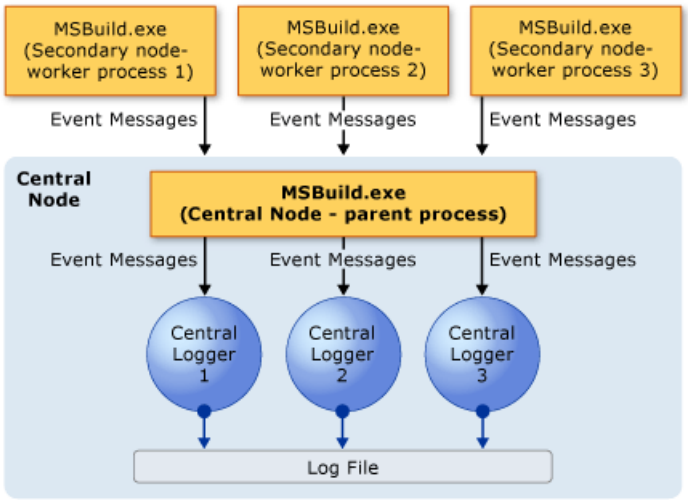
You can improve logging efficiency even more by creating a custom forwarding logger. A custom forwarding logger acts as a filter by letting you choose, before you build, only the events you want to monitor. When you use a custom forwarding logger, unwanted events cannot overwhelm the logger, clutter your logs, or slow build times.

Multi-processor logging models

To provide for multi-processor-related build issues, MSBuild supports two logging models, central and distributed.

Central logging model

In the central logging model, a single instance of *MSBuild.exe* acts as the "central node," and child instances of the central node ("secondary nodes") attach to the central node to help it perform build tasks.



Loggers of various types that attach to the central node are known as "central loggers." Only one instance of each logger type can be attached to the central node at the same time.

When a build occurs, the secondary nodes route their build events to the central node. The central node routes all its events, and also those of the secondary nodes, to one or more of the attached central loggers. The loggers then create log files that are based on the incoming data.

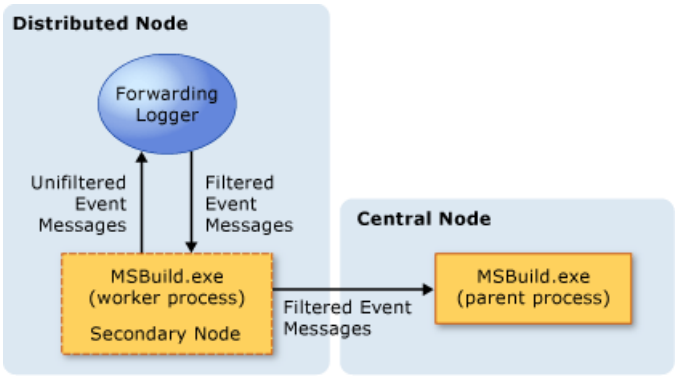
Although only ILogger is required to be implemented by the central logger, we recommend that you also implement INodeLogger so that the central logger initializes with the number of nodes that are participating in the build. The following overload of the Initialize method invokes when the engine initializes the logger.

```
public interface INodeLogger: ILogger
{
    public void Initialize(IEventSource eventSource, int nodeCount);
}
```

Any pre-existing ILogger-based loggers can act as central loggers and can attach to the build. However, central loggers written without explicit support for multi-processor logging scenarios and out-of-order events may break a build or produce meaningless output.

Distributed logging model

In the central logging model, too much incoming message traffic can overwhelm the central node, for example, when many projects build at the same time. This can stress system resources and decrease build performance. To ease this problem, MSBuild supports a distributed logging model.



The distributed logging model extends the central logging model by letting you create a forwarding logger.

Forwarding loggers

A forwarding logger is a secondary, lightweight logger that has an event filter that attaches to a secondary node and receives incoming build events from that node. It filters the incoming events and forwards only the ones that you specify to the central node. This reduces the message traffic that is sent to the central node and improves overall build performance.

There are two ways to use distributed logging, as follows:

- Customize the pre-fabricated forwarding logger named ConfigurableForwardingLogger.
- Write your own custom forwarding logger.

You can modify ConfigurableForwardingLogger to suit your requirements. To do this, call the logger on the command line by using *MSBuild.exe*, and list the build events that you want the logger to forward to the central node.

As an alternative, you can create a custom forwarding logger. By creating a custom forwarding logger, you can fine-tune the behavior of the logger. However, creating a custom forwarding logger is more complex than just customizing the ConfigurableForwardingLogger. For more information, see Creating forwarding loggers.

Using the ConfigurableForwardingLogger for simple distributed logging

To attach either a ConfigurableForwardingLogger or a custom forwarding logger, use the `-distributedlogger` switch (`-dl` for short) in an *MSBuild.exe* command-line build. The format for specifying the names of the logger

types and classes is the same as that for the `-logger` switch, except that a distributed logger always has two logging classes instead of one, the forwarding logger and the central logger. The following is an example of how to attach a custom forwarding logger named XMLForwardingLogger.

```
msbuild.exe myproj.proj -
distributedlogger:XMLCentralLogger,MyLogger,Version=1.0.2,Culture=neutral*XMLForwardingLogger,MyLogger,Version
=1.0.2,Culture=neutral
```

> **NOTE**
> An asterisk (*) must separate the two logger names in the `-dl` switch.

Using the ConfigurableForwardingLogger is like using any other logger (as outlined in Obtaining build logs), except that you attach the ConfigurableForwardingLogger logger instead of the typical MSBuild logger and you specify as parameters the events that you want the ConfigurableForwardingLogger to pass on to the central node.

For example, if you want to be notified only when a build starts and ends, and when an error occurs, you would pass `BUILDSTARTEDEVENT`, `BUILDFINISHEDEVENT`, and `ERROREVENT` as parameters. Multiple parameters can be passed by separating them with semi-colons. The following is an example of how to use the ConfigurableForwardingLogger to forward only the `BUILDSTARTEDEVENT`, `BUILDFINISHEDEVENT`, and `ERROREVENT` events.

```
msbuild.exe myproj.proj -
distributedlogger:XMLCentralLogger,MyLogger,Version=1.0.2,Culture=neutral*ConfigureableForwardingLogger,C:\My.
dll;BUILDSTARTEDEVENT; BUILDFINISHEDEVENT;ERROREVENT
```

The following is a list of the available ConfigurableForwardingLogger parameters.

CONFIGURABLEFORWARDINGLOGGER PARAMETERS
BUILDSTARTEDEVENT
BUILDFINISHEDEVENT
PROJECTSTARTEDEVENT
PROJECTFINISHEDEVENT
TARGETSTARTEDEVENT
TARGETFINISHEDEVENT
TASKSTARTEDEVENT
TASKFINISHEDEVENT
ERROREVENT
WARNINGEVENT
HIGHMESSAGEEVENT

CONFIGURABLEFORWARDINGLOGGER PARAMETERS
NORMALMESSAGEEVENT
LOWMESSAGEEVENT
CUSTOMEVENT
COMMANDLINE
PERFORMANCESUMMARY
NOSUMMARY
SHOWCOMMANDLINE

See also

- Creating forwarding loggers

Create forwarding loggers

2/21/2019 • 2 minutes to read • Edit Online

Forwarding loggers improve logging efficiency by letting you choose the events you want to monitor when you build projects on a multi-processor system. By enabling forwarding loggers, you can prevent unwanted events from overwhelming the central logger, slowing build time, and cluttering your log.

To create a forwarding logger, you can either implement the IForwardingLogger interface and then implement its methods manually, or use the ConfigurableForwardingLogger class and its pre-configured methods. (The latter will suffice for most applications.)

Register events and respond to them

A forwarding logger gathers information about build events as they are reported by the secondary build engine, which is a worker process that is created by the main build process during a build on a multi-processor system. Then the forwarding logger selects events to forward to the central logger, based on the instructions you have given it.

You must register forwarding loggers to handle the events you want to monitor. To register for events, loggers must override the Initialize method. This method now includes an optional parameter, `nodecount`, that can be set to the number of processors in the system. (By default, the value is 1.)

Examples of events you can monitor are TargetStarted, ProjectStarted, and ProjectFinished.

In a multi-processor environment, event messages are likely to be received out of order. Therefore, you must evaluate the events by using the event handler in the forwarding logger and program it to determine which events to pass to the redirector for forwarding to the central logger. To accomplish this, you can use the BuildEventContext class, which is attached to every message, to help identify events you want to forward, and then pass the names of the events to the ConfigurableForwardingLogger class (or a subclass of it). When you use this method, no other specific coding is required to forward events.

Specify a forwarding logger

After the forwarding logger has been compiled into an assembly, you must tell MSBuild to use it during builds. To do this, use the `-FileLogger`, `-FileLoggerParameters`, and `-DistributedFileLogger` switches together with *MSBuild.exe*. The `-FileLogger` switch tells *MSBuild.exe* that the logger is directly attached. The `-DistributedFileLogger` switch means that there is a log file per node. To set parameters on the forwarding logger, use the `-FileLoggerParameters` switch. For more information about these and other *MSBuild.exe* switches, see Command-line reference.

Multi-processor-aware loggers

When you build a project on a multi-processor system, the build messages from each processor are not automatically interleaved in a unified sequence. Instead, you must establish a message grouping priority by using the BuildEventContext class that is attached to every message. For more information about multi-processor building, see Logging in a multi-processor environment.

See also

- Obtain build logs
- Build loggers

- Logging in a multi-processor environment

Walkthrough: Use MSBuild

4/23/2019 • 14 minutes to read • Edit Online

MSBuild is the build platform for Microsoft and Visual Studio. This walkthrough introduces you to the building blocks of MSBuild and shows you how to write, manipulate, and debug MSBuild projects. You will learn about:

- Creating and manipulating a project file.
- How to use build properties
- How to use build items.

You can run MSBuild from Visual Studio, or from the **Command Window**. In this walkthrough, you create an MSBuild project file using Visual Studio. You edit the project file in Visual Studio, and use the **Command Window** to build the project and examine the results.

Create an MSBuild project

The Visual Studio project system is based on MSBuild. This makes it easy to create a new project file using Visual Studio. In this section, you create a Visual C# project file. You can choose to create a Visual Basic project file instead. In the context of this walkthrough, the difference between the two project files is minor.

To create a project file

1. Open Visual Studio and create a project.

   Press **Esc** to close the start window. Type **Ctrl + Q** to open the search box, type **winforms**, then choose **Create a new Windows Forms App (.NET Framework)**. In the dialog box that appears, choose **Create**.

   In the **Name** box, type `BuildApp`. Enter a **Location** for the solution, for example, *D*:\. Accept the defaults for **Solution**, **Solution Name** (**BuildApp**), and **Framework**.

   From the top menu bar, choose **File** > **New** > **Project**. In the left pane of the **New Project** dialog box, expand **Visual C#** > **Windows Desktop**, then choose **Windows Forms App (.NET Framework)**. Then choose **OK**.

   In the **Name** box, type `BuildApp`. Enter a **Location** for the solution, for example, *D*:\. Accept the defaults for **Create directory for solution** (selected), **Add to Source Control** (not selected), and **Solution Name** (**BuildApp**).

2. Click **OK** or **Create** to create the project file.

Examine the project file

In the previous section, you used Visual Studio to create a Visual C# project file. The project file is represented in **Solution Explorer** by the project node named BuildApp. You can use the Visual Studio code editor to examine the project file.

To examine the project file

1. In **Solution Explorer**, click the project node **BuildApp**.
2. In the **Properties** browser, notice that the **Project File** property is *BuildApp.csproj*. All project files are named with the suffix *proj*. If you had created a Visual Basic project, the project file name would be *BuildApp.vbproj*.

3. Right-click the project node, then click **Unload Project**.

4. Right-click the project node again, then click **Edit BuildApp.csproj**.

   The project file appears in the code editor.

Targets and tasks

Project files are XML-formatted files with the root node Project.

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="15.0"  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
```

You must specify the xmlns namespace in the Project element. If `ToolsVersion` is present in a new project, it must be "15.0".

The work of building an application is done with Target and Task elements.

- A task is the smallest unit of work, in other words, the "atom" of a build. Tasks are independent executable components which may have inputs and outputs. There are no tasks currently referenced or defined in the project file. You add tasks to the project file in the sections below. For more information, see the Tasks topic.
- A target is a named sequence of tasks. For more information, see the Targets topic.

The default target is not defined in the project file. Instead, it is specified in imported projects. The Import element specifies imported projects. For example, in a C# project, the default target is imported from the file *Microsoft.CSharp.targets*.

```
<Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />
```

Imported files are effectively inserted into the project file wherever they are referenced.

> **NOTE**
> Some project types, such as .NET Core, use a simplified schema with an `Sdk` attribute instead of `ToolsVersion`. These projects have implicit imports and different default attribute values.

MSBuild keeps track of the targets of a build, and guarantees that each target is built no more than once.

Add a target and a task

Add a target to the project file. Add a task to the target that prints out a message.

To add a target and a task

1. Add these lines to the project file, just after the Import statement:

   ```
   <Target Name="HelloWorld">
   </Target>
   ```

   This creates a target named HelloWorld. Notice that you have IntelliSense support while editing the project file.

2. Add lines to the HelloWorld target, so that the resulting section looks like this:

```
<Target Name="HelloWorld">
  <Message Text="Hello"></Message>  <Message Text="World"></Message>
</Target>
```

3. Save the project file.

The Message task is one of the many tasks that ships with MSBuild. For a complete list of available tasks and usage information, see Task reference.

The Message task takes the string value of the Text attribute as input and displays it on the output device. The HelloWorld target executes the Message task twice: first to display "Hello", and then to display "World".

Build the target

Run MSBuild from the **Developer Command Prompt** for Visual Studio to build the HelloWorld target defined above. Use the -target or -t command line switch to select the target.

> **NOTE**
> We will refer to the **Developer Command Prompt** as the **Command Window** in the sections below.

To build the target

1. Open the **Command Window**.

   (Windows 10) In the search box on the taskbar, start typing the name of the tool, such as `dev` or `developer command prompt`. This brings up a list of installed apps that match your search pattern.

   If you need to find it manually, the file is *LaunchDevCmd.bat* in the *<visualstudio installation folder>\<version>\Common7\Tools* folder.

2. From the command window, navigate to the folder containing the project file, in this case, *D:\BuildApp\BuildApp*.

3. Run msbuild with the command switch -t:HelloWorld. This selects and builds the HelloWorld target:

```
msbuild buildapp.csproj -t:HelloWorld
```

4. Examine the output in the **Command window**. You should see the two lines "Hello" and "World":

```
Hello
World
```

> **NOTE**
> If instead you see `The target "HelloWorld" does not exist in the project` then you probably forgot to save the project file in the code editor. Save the file and try again.

By alternating between the code editor and the command window, you can change the project file and quickly see the results.

Build properties

Build properties are name-value pairs that guide the build. Several build properties are already defined at the top of the project file:

```
<PropertyGroup>
...
  <ProductVersion>10.0.11107</ProductVersion>
  <SchemaVersion>2.0</SchemaVersion>
  <ProjectGuid>{30E3C9D5-FD86-4691-A331-80EA5BA7E571}</ProjectGuid>
  <OutputType>WinExe</OutputType>
...
</PropertyGroup>
```

All properties are child elements of PropertyGroup elements. The name of the property is the name of the child element, and the value of the property is the text element of the child element. For example,

```
<TargetFrameworkVersion>v15.0</TargetFrameworkVersion>
```

defines the property named TargetFrameworkVersion, giving it the string value "v15.0".

Build properties may be redefined at any time. If

```
<TargetFrameworkVersion>v3.5</TargetFrameworkVersion>
```

appears later in the project file, or in a file imported later in the project file, then TargetFrameworkVersion takes the new value "v3.5".

Examine a property value

To get the value of a property, use the following syntax, where PropertyName is the name of the property:

```
$(PropertyName)
```

Use this syntax to examine some of the properties in the project file.

To examine a property value

1. From the code editor, replace the HelloWorld target with this code:

   ```
   <Target Name="HelloWorld">
     <Message Text="Configuration is $(Configuration)" />
     <Message Text="MSBuildToolsPath is $(MSBuildToolsPath)" />
   </Target>
   ```

2. Save the project file.

3. From the **Command Window**, enter and execute this line:

   ```
   msbuild buildapp.csproj -t:HelloWorld
   ```

4. Examine the output. You should see these two lines (your .NET Framework version may differ):

```
Configuration is Debug
MSBuildToolsPath is C:\Program Files (x86)\Microsoft Visual Studio\2019\<Visual Studio
SKU>\MSBuild\15.0\Bin
```

```
Configuration is Debug
MSBuildToolsPath is C:\Program Files (x86)\Microsoft Visual Studio\2017\<Visual Studio
SKU>\MSBuild\15.0\Bin
```

> **NOTE**
> If you don't see these lines then you probably forgot to save the project file in the code editor. Save the file and try again.

Conditional properties

Many properties like Configuration are defined conditionally, that is, the Condition attribute appears in the property element. Conditional properties are defined or redefined only if the condition evaluates to "true". Note that undefined properties are given the default value of an empty string. For example,

```
<Configuration   Condition=" '$(Configuration)' == '' ">Debug</Configuration>
```

means "If the Configuration property has not been defined yet, define it and give it the value 'Debug'".

Almost all MSBuild elements can have a Condition attribute. For more discussion about using the Condition attribute, see Conditions.

Reserved properties

MSBuild reserves some property names to store information about the project file and the MSBuild binaries. MSBuildToolsPath is an example of a reserved property. Reserved properties are referenced with the $ notation like any other property. For more information, see How to: Reference the name or location of the project file and MSBuild reserved and well-known properties.

Environment variables

You can reference environment variables in project files the same way as build properties. For example, to use the PATH environment variable in your project file, use $(Path). If the project contains a property definition that has the same name as an environment variable, the property in the project overrides the value of the environment variable. For more information, see How to: Use environment variables in a build.

Set properties from the command line

Properties may be defined on the command line using the -property or -p command line switch. Property values received from the command line override property values set in the project file and environment variables.

To set a property value from the command line

1. From the **Command Window**, enter and execute this line:

```
msbuild buildapp.csproj -t:HelloWorld -p:Configuration=Release
```

2. Examine the output. You should see this line:

```
Configuration is Release.
```

MSBuild creates the Configuration property and gives it the value "Release".

Special characters

Certain characters have special meaning in MSBuild project files. Examples of these characters include semicolons (;) and asterisks (*). In order to use these special characters as literals in a project file, they must be specified by using the syntax %<xx>, where <xx> represents the ASCII hexadecimal value of the character.

Change the Message task to show the value of the Configuration property with special characters to make it more readable.

To use special characters in the Message task

1. From the code editor, replace both Message tasks with this line:

```
<Message Text="%24(Configuration) is %22$(Configuration)%22" />
```

2. Save the project file.

3. From the **Command Window**, enter and execute this line:

```
msbuild buildapp.csproj -t:HelloWorld
```

4. Examine the output. You should see this line:

```
$(Configuration) is "Debug"
```

For more information, see MSBuild special characters.

Build items

An item is a piece of information, typically a file name, that is used as an input to the build system. For example, a collection of items representing source files might be passed to a task named Compile to compile them into an assembly.

All items are child elements of ItemGroup elements. The item name is the name of the child element, and the item value is the value of the Include attribute of the child element. The values of items with the same name are collected into item types of that name. For example,

```
<ItemGroup>
    <Compile Include="Program.cs" />
    <Compile Include="Properties\AssemblyInfo.cs" />
</ItemGroup>
```

defines an item group containing two items. The item type Compile has two values: *Program.cs* and *Properties\AssemblyInfo.cs*.

The following code creates the same item type by declaring both files in one Include attribute, separated by a semicolon.

```
<ItemGroup>
    <Compile Include="Program.cs;Properties\AssemblyInfo.cs" />
</ItemGroup>
```

For more information, see Items.

> **NOTE**
> File paths are relative to the folder containing the MSBuild project file.

Examine item type values

To get the values of an item type, use the following syntax, where ItemType is the name of the item type:

```
@(ItemType)
```

Use this syntax to examine the Compile item type in the project file.

To examine item type values

1. From the code editor, replace the HelloWorld target task with this code:

   ```
   <Target Name="HelloWorld">
     <Message Text="Compile item type contains @(Compile)" />
   </Target>
   ```

2. Save the project file.

3. From the **Command Window**, enter and execute this line:

   ```
   msbuild buildapp.csproj -t:HelloWorld
   ```

4. Examine the output. You should see this long line:

   ```
   Compile item type contains
   Form1.cs;Form1.Designer.cs;Program.cs;Properties\AssemblyInfo.cs;Properties\Resources.Designer.cs;Prope
   rties\Settings.Designer.cs
   ```

The values of an item type are separated with semicolons by default.

To change the separator of an item type, use the following syntax, where ItemType is the item type and Separator is a string of one or more separating characters:

```
@(ItemType, Separator)
```

Change the Message task to use carriage returns and line feeds (%0A%0D) to display Compile items one per line.

To display item type values one per line

1. From the code editor, replace the Message task with this line:

   ```
   <Message Text="Compile item type contains @(Compile, '%0A%0D')" />
   ```

2. Save the project file.

3. From the **Command Window**, enter and execute this line:

```
msbuild buildapp.csproj -t:HelloWorld
```

4. Examine the output. You should see these lines:

```
Compile item type contains Form1.cs
Form1.Designer.cs
Program.cs
Properties\AssemblyInfo.cs
Properties\Resources.Designer.cs
Properties\Settings.Designer.cs
```

Include, Exclude, and wildcards

You can use the wildcards "*", "**", and "?" with the Include attribute to add items to an item type. For example,

```
<Photos Include="images\*.jpeg" />
```

adds all files with the file extension *.jpeg* in the *images* folder to the Photos item type, while

```
<Photos Include="images\**\*.jpeg" />
```

adds all files with the file extension *.jpeg* in the *images* folder, and all its subfolders, to the Photos item type. For more examples, see How to: Select the files to build.

Notice that as items are declared they are added to the item type. For example,

```
<Photos Include="images\*.jpeg" />
<Photos Include="images\*.gif" />
```

creates an item type named Photo containing all files in the *images* folder with a file extension of either *.jpeg* or *.gif*. This is equivalent to the following line:

```
<Photos Include="images\*.jpeg;images\*.gif" />
```

You can exclude an item from an item type with the Exclude attribute. For example,

```
<Compile Include="*.cs" Exclude="*Designer*">
```

adds all files with the file extension *.cs* to the Compile item type, except for files whose names contain the string *Designer*. For more examples, see How to: Exclude files from the build.

The Exclude attribute only affects the items added by the Include attribute in the item element that contains them both. For example,

```
<Compile Include="*.cs" />
<Compile Include="*.res" Exclude="Form1.cs">
```

would not exclude the file *Form1.cs*, which was added in the preceding item element.

To include and exclude items

1. From the code editor, replace the Message task with this line:

```
<Message Text="XFiles item type contains @(XFiles)" />
```

2. Add this item group just after the Import element:

```
<ItemGroup>
   <XFiles Include="*.cs;properties/*.resx" Exclude="*Designer*" />
</ItemGroup>
```

3. Save the project file.

4. From the **Command Window**, enter and execute this line:

```
msbuild buildapp.csproj -t:HelloWorld
```

5. Examine the output. You should see this line:

```
XFiles item type contains Form1.cs;Program.cs;Properties/Resources.resx
```

Item metadata

Items may contain metadata in addition to the information gathered from the Include and Exclude attributes. This metadata can be used by tasks that require more information about items than just the item value.

Item metadata is declared in the project file by creating an element with the name of the metadata as a child element of the item. An item can have zero or more metadata values. For example, the following CSFile item has Culture metadata with a value of "Fr":

```
<ItemGroup>
    <CSFile Include="main.cs">
        <Culture>Fr</Culture>
    </CSFile>
</ItemGroup>
```

To get the metadata value of an item type, use the following syntax, where ItemType is the name of the item type and MetaDataName is the name of the metadata:

```
%(ItemType.MetaDataName)
```

To examine item metadata

1. From the code editor, replace the Message task with this line:

```
<Message Text="Compile.DependentUpon: %(Compile.DependentUpon)" />
```

2. Save the project file.

3. From the **Command Window**, enter and execute this line:

```
msbuild buildapp.csproj -t:HelloWorld
```

4. Examine the output. You should see these lines:

```
Compile.DependentUpon:
Compile.DependentUpon: Form1.cs
Compile.DependentUpon: Resources.resx
Compile.DependentUpon: Settings.settings
```

Notice how the phrase "Compile.DependentUpon" appears several times. The use of metadata with this syntax within a target causes "batching". Batching means that the tasks within the target are executed once for each unique metadata value. This is the MSBuild script equivalent of the common "for loop" programming construct. For more information, see Batching.

Well-known metadata

Whenever an item is added to an item list, that item is assigned some well-known metadata. For example, %(Filename) returns the file name of any item. For a complete list of well-known metadata, see Well-known item metadata.

To examine well-known metadata

1. From the code editor, replace the Message task with this line:

```
<Message Text="Compile Filename: %(Compile.Filename)" />
```

2. Save the project file.

3. From the **Command Window**, enter and execute this line:

```
msbuild buildapp.csproj -t:HelloWorld
```

4. Examine the output. You should see these lines:

```
Compile Filename: Form1
Compile Filename: Form1.Designer
Compile Filename: Program
Compile Filename: AssemblyInfo
Compile Filename: Resources.Designer
Compile Filename: Settings.Designer
```

By comparing the two examples above, you can see that while not every item in the Compile item type has DependentUpon metadata, all items have the well-known Filename metadata.

Metadata transformations

Item lists can be transformed into new item lists. To transform an item list, use the following syntax, where <ItemType> is the name of the item type and <MetadataName> is the name of the metadata:

```
@(ItemType -> '%(MetadataName)')
```

For example, an item list of source files can be transformed into a collection of object files using an expression like `@(SourceFiles -> '%(Filename).obj')`. For more information, see Transforms.

To transform items using metadata

1. From the code editor, replace the Message task with this line:

```
<Message Text="Backup files: @(Compile->'%(filename).bak')" />
```

2. Save the project file.
3. From the **Command Window**, enter and execute this line:

```
msbuild buildapp.csproj -t:HelloWorld
```

4. Examine the output. You should see this line:

```
Backup files:
Form1.bak;Form1.Designer.bak;Program.bak;AssemblyInfo.bak;Resources.Designer.bak;Settings.Designer.bak
```

Notice that metadata expressed in this syntax does not cause batching.

What's next?

To learn how to create a simple project file one step at a time, try out the Walkthrough: Creating an MSBuild project file from scratch.

See also

- MSBuild overview
- MSBuild reference

Walkthrough: Create an MSBuild project file from scratch

10/21/2019 • 10 minutes to read • Edit Online

Programming languages that target the .NET Framework use MSBuild project files to describe and control the application build process. When you use Visual Studio to create an MSBuild project file, the appropriate XML is added to the file automatically. However, you may find it helpful to understand how the XML is organized and how you can change it to control a build.

For information about creating a project file for a C++ project, see MSBuild (C++).

This walkthrough shows how to create a basic project file incrementally, by using only a text editor. The walkthrough follows these steps:

1. Create a minimal application source file.
2. Create a minimal MSBuild project file.
3. Extend the PATH environment variable to include MSBuild.
4. Build the application by using the project file.
5. Add properties to control the build.
6. Control the build by changing property values.
7. Add targets to the build.
8. Control the build by specifying targets.
9. Build incrementally.

This walkthrough shows how to build the project at the command prompt and examine the results. For more information about MSBuild and how to run MSBuild at the command prompt, see Walkthrough: Use MSBuild.

To complete the walkthrough, you must have the .NET Framework (version 2.0, 3.5, 4.0, or 4.5) installed because it includes MSBuild and the Visual C# compiler, which are required for the walkthrough.

Create a minimal application

This section shows how to create a minimal Visual C# application source file by using a text editor.

To create the minimal application

1. At the command prompt, browse to the folder where you want to create the application, for example, *My Documents*\ or *Desktop*\.
2. Type **md HelloWorld** to create a subfolder named *HelloWorld*\.
3. Type **cd HelloWorld** to change to the new folder.
4. Start Notepad or another text editor, and then type the following code.

```
using System;

class HelloWorld
{
    static void Main()
    {
#if DebugConfig
        Console.WriteLine("WE ARE IN THE DEBUG CONFIGURATION");
#endif

        Console.WriteLine("Hello, world!");
    }
}
```

5. Save this source code file and name it *Helloworld.cs*.
6. Build the application by typing **csc helloworld.cs** at the command prompt.
7. Test the application by typing **helloworld** at the command prompt.

   The **Hello, world!** message should be displayed.
8. Delete the application by typing **del helloworld.exe** at the command prompt.

Create a minimal MSBuild project file

Now that you have a minimal application source file, you can create a minimal project file to build the application. This project file contains the following elements:

- The required root `Project` node.
- An `ItemGroup` node to contain item elements.
- An item element that refers to the application source file.
- A `Target` node to contain tasks that are required to build the application.
- A `Task` element to start the Visual C# compiler to build the application.

To create a minimal MSBuild project file

1. In the text editor, replace the existing text by using these two lines:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
</Project>
```

2. Insert this `ItemGroup` node as a child element of the `Project` node:

```
<ItemGroup>
  <Compile Include="helloworld.cs" />
</ItemGroup>
```

   Notice that this `ItemGroup` already contains an item element.

3. Add a `Target` node as a child element of the `Project` node. Name the node `Build`.

```
<Target Name="Build">
</Target>
```

4. Insert this task element as a child element of the `Target` node:

```
<Csc Sources="@(Compile)"/>
```

5. Save this project file and name it *Helloworld.csproj*.

Your minimal project file should resemble the following code:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <ItemGroup>
    <Compile Include="helloworld.cs" />
  </ItemGroup>
  <Target Name="Build">
    <Csc Sources="@(Compile)"/>
  </Target>
</Project>
```

Tasks in the Build target are executed sequentially. In this case, the Visual C# compiler `Csc` task is the only task. It expects a list of source files to compile, and this is given by the value of the `Compile` item. The `Compile` item references just one source file, *Helloworld.cs*.

> **NOTE**
>
> In the item element, you can use the asterisk wildcard character (*) to reference all files that have the *.cs* file name extension, as follows:
>
> ```
> <Compile Include="*.cs" />
> ```
>
> However, we do not recommend the use of wildcard characters because it makes debugging and selective targeting more difficult if source files are added or deleted.

Extend the path to include MSBuild

Before you can access MSBuild, you must extend the PATH environment variable to include the .NET Framework folder.

To add MSBuild to your path

- Starting in Visual Studio 2013, you can find *MSBuild.exe* in the MSBuild folder (*%ProgramFiles%\MSBuild* on a 32-bit operating system, or *%ProgramFiles(x86)%\MSBuild* on a 64-bit operating system).

  At the command prompt, type **set PATH=%PATH%;%ProgramFiles%\MSBuild** or **set PATH=%PATH%;%ProgramFiles(x86)%\MSBuild**.

  Alternatively, if you have Visual Studio installed, you can use the **Visual Studio Command Prompt**, which has a path that includes the *MSBuild* folder.

Use the project file to build the application

Now, to build the application, use the project file that you just created.

To build the application

1. At the command prompt, type **msbuild helloworld.csproj -t:Build**.

   This builds the Build target of the Helloworld project file by invoking the Visual C# compiler to create the Helloworld application.

2. Test the application by typing **helloworld**.

   The **Hello, world!** message should be displayed.

> **NOTE**
> You can see more details about the build by increasing the verbosity level. To set the verbosity level to "detailed", type this command at the command prompt:
>
> **msbuild helloworld.csproj -t:Build -verbosity:detailed**

Add build properties

You can add build properties to the project file to further control the build. Now add these properties:

- An `AssemblyName` property to specify the name of the application.
- An `OutputPath` property to specify a folder to contain the application.

To add build properties

1. Delete the existing application by typing **del helloworld.exe** at the command prompt.
2. In the project file, insert this `PropertyGroup` element just after the opening `Project` element:

   ```
   <PropertyGroup>
     <AssemblyName>MSBuildSample</AssemblyName>
     <OutputPath>Bin\</OutputPath>
   </PropertyGroup>
   ```

3. Add this task to the Build target, just before the `Csc` task:

   ```
   <MakeDir Directories="$(OutputPath)"     Condition="!Exists('$(OutputPath)')" />
   ```

   The `MakeDir` task creates a folder that is named by the `OutputPath` property, provided that no folder by that name currently exists.

4. Add this `OutputAssembly` attribute to the `Csc` task:

   ```
   <Csc Sources="@(Compile)" OutputAssembly="$(OutputPath)$(AssemblyName).exe" />
   ```

   This instructs the Visual C# compiler to produce an assembly that is named by the `AssemblyName` property and to put it in the folder that is named by the `OutputPath` property.

5. Save your changes.

Your project file should now resemble the following code:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <AssemblyName>MSBuildSample</AssemblyName>
    <OutputPath>Bin\</OutputPath>
  </PropertyGroup>
  <ItemGroup>
    <Compile Include="helloworld.cs" />
  </ItemGroup>
  <Target Name="Build">
    <MakeDir Directories="$(OutputPath)" Condition="!Exists('$(OutputPath)')" />
    <Csc Sources="@(Compile)" OutputAssembly="$(OutputPath)$(AssemblyName).exe" />
  </Target>
</Project>
```

NOTE

We recommend that you add the backslash (\) path delimiter at the end of the folder name when you specify it in the `OutputPath` element, instead of adding it in the `OutputAssembly` attribute of the `Csc` task. Therefore,

`<OutputPath>Bin\</OutputPath>`

`OutputAssembly=="$(OutputPath)$(AssemblyName).exe" />`

is better than

`<OutputPath>Bin</OutputPath>`

`OutputAssembly=="$(OutputPath)\$(AssemblyName).exe" />`

Test the build properties

Now you can build the application by using the project file in which you used build properties to specify the output folder and application name.

To test the build properties

1. At the command prompt, type **msbuild helloworld.csproj -t:Build**.

   This creates the *\Bin* folder and then invokes the Visual C# compiler to create the *MSBuildSample* application and puts it in the *\Bin* folder.

2. To verify that the *\Bin* folder has been created, and that it contains the *MSBuildSample* application, type **dir Bin**.

3. Test the application by typing **Bin\MSBuildSample**.

   The **Hello, world!** message should be displayed.

Add build targets

Next, add two more targets to the project file, as follows:

- A Clean target that deletes old files.
- A Rebuild target that uses the `DependsOnTargets` attribute to force the Clean task to run before the Build task.

Now that you have multiple targets, you can set the Build target as the default target.

To add build targets

1. In the project file, add these two targets just after the Build target:

```
<Target Name="Clean" >
  <Delete Files="$(OutputPath)$(AssemblyName).exe" />
</Target>
<Target Name="Rebuild" DependsOnTargets="Clean;Build" />
```

The Clean target invokes the Delete task to delete the application. The Rebuild target does not run until both the Clean target and the Build target have run. Although the Rebuild target has no tasks, it causes the Clean target to run before the Build target.

2. Add this `DefaultTargets` attribute to the opening `Project` element:

```
<Project DefaultTargets="Build" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
```

This sets the Build target as the default target.

Your project file should now resemble the following code:

```
<Project DefaultTargets="Build" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <AssemblyName>MSBuildSample</AssemblyName>
    <OutputPath>Bin\</OutputPath>
  </PropertyGroup>
  <ItemGroup>
    <Compile Include="helloworld.cs" />
  </ItemGroup>
  <Target Name="Build">
    <MakeDir Directories="$(OutputPath)" Condition="!Exists('$(OutputPath)')" />
    <Csc Sources="@(Compile)" OutputAssembly="$(OutputPath)$(AssemblyName).exe" />
  </Target>
  <Target Name="Clean" >
    <Delete Files="$(OutputPath)$(AssemblyName).exe" />
  </Target>
  <Target Name="Rebuild" DependsOnTargets="Clean;Build" />
</Project>
```

Test the build targets

You can exercise the new build targets to test these features of the project file:

- Building the default build.
- Setting the application name at the command prompt.
- Deleting the application before another application is built.
- Deleting the application without building another application.

To test the build targets

1. At the command prompt, type **msbuild helloworld.csproj -p:AssemblyName=Greetings**.

   Because you did not use the **-t** switch to explicitly set the target, MSBuild runs the default Build target. The **-p** switch overrides the `AssemblyName` property and gives it the new value, `Greetings`. This causes a new application, *Greetings.exe*, to be created in the *\Bin* folder.

2. To verify that the *\Bin* folder contains both the *MSBuildSample* application and the new *Greetings* application, type **dir Bin**.

3. Test the Greetings application by typing **Bin\Greetings**.

The **Hello, world!** message should be displayed.

4. Delete the MSBuildSample application by typing **msbuild helloworld.csproj -t:clean**.

   This runs the Clean task to remove the application that has the default `AssemblyName` property value, `MSBuildSample`.

5. Delete the Greetings application by typing **msbuild helloworld.csproj -t:clean -p:AssemblyName=Greetings**.

   This runs the Clean task to remove the application that has the given **AssemblyName** property value, `Greetings`.

6. To verify that the *Bin*\ folder is now empty, type **dir Bin**.

7. Type **msbuild**.

   Although a project file is not specified, MSBuild builds the *helloworld.csproj* file because there is only one project file in the current folder. This causes the *MSBuildSample* application to be created in the *Bin*\ folder.

   To verify that the *Bin*\ folder contains the *MSBuildSample* application, type **dir Bin**.

Build incrementally

You can tell MSBuild to build a target only if the source files or target files that the target depends on have changed. MSBuild uses the time stamp of a file to determine whether it has changed.

To build incrementally

1. In the project file, add these attributes to the opening Build target:

   ```
   Inputs="@(Compile)" Outputs="$(OutputPath)$(AssemblyName).exe"
   ```

   This specifies that the Build target depends on the input files that are specified in the `Compile` item group, and that the output target is the application file.

   The resulting Build target should resemble the following code:

   ```
   <Target Name="Build" Inputs="@(Compile)" Outputs="$(OutputPath)$(AssemblyName).exe">
     <MakeDir Directories="$(OutputPath)" Condition="!Exists('$(OutputPath)')" />
     <Csc Sources="@(Compile)" OutputAssembly="$(OutputPath)$(AssemblyName).exe" />
   </Target>
   ```

2. Test the Build target by typing **msbuild -v:d** at the command prompt.

   Remember that *helloworld.csproj* is the default project file, and that Build is the default target.

   The **-v:d** switch specifies a verbose description for the build process.

   These lines should be displayed:

   Skipping target "Build" because all output files are up-to-date with respect to the input files.

   Input files: HelloWorld.cs

   Output files: BinMSBuildSample.exe

   MSBuild skips the Build target because none of the source files have changed since the application was last built.

Example

Description

The following example shows a project file that compiles a Visual C# application and logs a message that contains the output file name.

Code

```
<Project DefaultTargets = "Compile"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003" >

    <!-- Set the application name as a property -->
    <PropertyGroup>
        <appname>HelloWorldCS</appname>
    </PropertyGroup>

    <!-- Specify the inputs by type and file name -->
    <ItemGroup>
        <CSFile Include = "consolehwcs1.cs"/>
    </ItemGroup>

    <Target Name = "Compile">
        <!-- Run the Visual C# compilation using input files of type CSFile -->
        <CSC
            Sources = "@(CSFile)"
            OutputAssembly = "$(appname).exe">
            <!-- Set the OutputAssembly attribute of the CSC task
            to the name of the executable file that is created -->
            <Output
                TaskParameter = "OutputAssembly"
                ItemName = "EXEFile" />
        </CSC>
        <!-- Log the file name of the output file -->
        <Message Text="The output file is @(EXEFile)"/>
    </Target>
</Project>
```

Example

Description

The following example shows a project file that compiles a Visual Basic application and logs a message that contains the output file name.

Code

```
<Project DefaultTargets = "Compile"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003" >

    <!-- Set the application name as a property -->
    <PropertyGroup>
        <appname>HelloWorldVB</appname>
    </PropertyGroup>

    <!-- Specify the inputs by type and file name -->
    <ItemGroup>
        <VBFile Include = "consolehwvb1.vb"/>
    </ItemGroup>

    <Target Name = "Compile">
        <!-- Run the Visual Basic compilation using input files of type VBFile -->
        <VBC
            Sources = "@(VBFile)"
            OutputAssembly= "$(appname).exe">
            <!-- Set the OutputAssembly attribute of the VBC task
            to the name of the executable file that is created -->
            <Output
                TaskParameter = "OutputAssembly"
                ItemName = "EXEFile" />
        </VBC>
        <!-- Log the file name of the output file -->
        <Message Text="The output file is @(EXEFile)"/>
    </Target>
</Project>
```

What's next?

Visual Studio can automatically do much of the work that is shown in this walkthrough. To learn how to use Visual Studio to create, edit, build, and test MSBuild project files, see Walkthrough: Use MSBuild.

See also

- MSBuild overview
- MSBuild reference

MSBuild reference

10/24/2019 • 2 minutes to read • Edit Online

MSBuild is the build system for Visual Studio. The following links lead to topics that contain MSBuild reference information.

In this section

- Project file schema reference

Describes the XML elements that make up the MSBuild file format.

- Task reference

Describes some of the typical tasks that are included with MSBuild.

- Conditions

Describes the conditions that are available in MSBuild files.

- Conditional constructs

Describes how to use the `Choose`, `When`, and `Otherwise` elements.

- MSBuild reserved and well-known properties

Describes the MSBuild reserved properties.

- Common MSBuild project properties

Describes project properties that are common to all project types, and also properties that are often used by particular project types.

- Common MSBuild project items

Describes project items that are common to all project types, and also items that are often used by particular project types.

- Command-line reference

Describes the arguments and switches that can be used with MSBuild.exe.

- .Targets files

Describes the *.Targets* file that is included in MSBuild.

- Well-known item metadata

Lists the metadata that is created together with every item.

- Response files

Explains the *.rsp* files that contain command-line switches.

- Additional resources Provides links to MSBuild websites and newsgroups.

- WPF MSBuild reference

Contains an MSBuild targets and task reference for Windows Presentation Foundation (WPF).

- Special characters to escape

Lists the characters that may have to be "escaped" to be interpreted correctly. An escape sequence is a series of characters that signifies that what follows is an alternative interpretation.

See also

- MSBuild overview
- Microsoft.Build.Evaluation
- Microsoft.Build.Execution
- Microsoft.Build.Framework
- Microsoft.Build.Logging
- Microsoft.Build.Tasks
- Microsoft.Build.Utilities

MSBuild project file schema reference

2/21/2019 • 2 minutes to read • Edit Online

Provides a table of all the MSBuild XML Schema elements with their available attributes and child elements.

MSBuild uses project files to instruct the build engine what to build and how to build it. MSBuild project files are XML files that adhere to the MSBuild XML schema. This section documents the XML schema definition (*.xsd*) file for MSBuild.

MSBuild XML schema elements

The following table lists all of the MSBuild XML schema elements along with their child elements and attributes.

ELEMENT	CHILD ELEMENTS	ATTRIBUTES
Choose element (MSBuild)	Otherwise When	--
Import element (MSBuild)	--	Condition Project
ImportGroup element	Import	Condition
Item element (MSBuild)	*ItemMetaData*	Condition Exclude Include Remove
ItemDefinitionGroup element (MSBuild)	*Item*	Condition
ItemGroup element (MSBuild)	*Item*	Condition
ItemMetadata element (MSBuild)	*Item*	Condition
OnError element (MSBuild)	--	Condition ExecuteTargets
Otherwise element (MSBuild)	Choose ItemGroup PropertyGroup	--

ELEMENT	CHILD ELEMENTS	ATTRIBUTES
Output element (MSBuild)	--	Condition ItemName PropertyName TaskParameter
Parameter element	--	Output ParameterType Required
ParameterGroup element	*Parameter*	--
Project element (MSBuild)	Choose Import ItemGroup ProjectExtensions PropertyGroup Target UsingTask	DefaultTargets InitialTargets ToolsVersion TreatAsLocalProperty xmlns
ProjectExtensions element (MSBuild)	--	--
Property element (MSBuild)	--	Condition
PropertyGroup element (MSBuild)	*Property*	Condition
Sdk element (MSBuild)	--	Name Version
Target element (MSBuild)	OnError *Task*	AfterTargets BeforeTargets Condition DependsOnTargets Inputs KeepDuplicateOutputs Name Outputs Returns

ELEMENT	CHILD ELEMENTS	ATTRIBUTES
Task element (MSBuild)	Output	Condition ContinueOnError *Parameter*
TaskBody element (MSBuild)	*Data*	Evaluate
UsingTask element (MSBuild)	ParameterGroup TaskBody	AssemblyFile AssemblyName Condition TaskFactory TaskName
When element (MSBuild)	Choose ItemGroup PropertyGroup	Condition

See also

- Task reference
- Conditions
- MSBuild reference
- MSBuild

Choose element (MSBuild)

2/21/2019 • 2 minutes to read • Edit Online

Evaluates child elements to select one set of `ItemGroup` elements and/or `PropertyGroup` elements to evaluate.

<Project> <Choose> <When> <Choose> ... <Otherwise> <Choose> ...

Syntax

```
<Choose>
    <When Condition="'StringA'=='StringB'">... </When>
    <Otherwise>... </Otherwise>
</Choose>
```

Attributes and elements

The following sections describe attributes, child elements, and parent elements.

Attributes

None.

Child elements

ELEMENT	DESCRIPTION
Otherwise	Optional element. Specifies the block of code `PropertyGroup` and `ItemGroup` elements to evaluate if the conditions of all `When` elements evaluate to `false`. There may be zero or one `Otherwise` elements in a `Choose` element, and it must be the last element.
When	Required element. Specifies a possible block of code for the `Choose` element to select. There may be one or more `When` elements in a `Choose` element.

Parent elements

ELEMENT	DESCRIPTION
Otherwise	Specifies the block of code to execute if the conditions of all `When` elements evaluate to `false`.
Project	Required root element of an MSBuild project file.
When	Specifies a possible block of code for the `Choose` element to select.

Remarks

The `Choose`, `When`, and `Otherwise` elements are used together to provide a way to select one section of code to execute out of a number of possible alternatives. For more information, see Conditional constructs.

Example

The following project uses the `Choose` element to select which set of property values in the `When` elements to set. If the `Condition` attributes of both `When` elements evaluate to `false`, the property values in the `Otherwise` element are set.

```
<Project
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003" >
    <PropertyGroup>
        <Configuration Condition="'$(Configuration)' == ''">Debug</Configuration>
        <OutputType>Exe</OutputType>
        <RootNamespace>ConsoleApplication1</RootNamespace>
        <AssemblyName>ConsoleApplication1</AssemblyName>
        <WarningLevel>4</WarningLevel>
    </PropertyGroup>
    <Choose>
        <When Condition=" '$(Configuration)'=='debug' ">
            <PropertyGroup>
                <DebugSymbols>true</DebugSymbols>
                <DebugType>full</DebugType>
                <Optimize>false</Optimize>
                <OutputPath>.\bin\Debug\</OutputPath>
                <DefineConstants>DEBUG;TRACE</DefineConstants>
            </PropertyGroup>
            <ItemGroup>
                <Compile Include="UnitTesting\*.cs" />
                <Reference Include="NUnit.dll" />
            </ItemGroup>
        </When>
        <When Condition=" '$(Configuration)'=='retail' ">
            <PropertyGroup>
                <DebugSymbols>false</DebugSymbols>
                <Optimize>true</Optimize>
                <OutputPath>.\bin\Release\</OutputPath>
                <DefineConstants>TRACE</DefineConstants>
            </PropertyGroup>
        </When>
        <Otherwise>
            <PropertyGroup>
                <DebugSymbols>true</DebugSymbols>
                <Optimize>false</Optimize>
                <OutputPath>.\bin\$(Configuration)\</OutputPath>
                <DefineConstants>DEBUG;TRACE</DefineConstants>
            </PropertyGroup>
        </Otherwise>
    </Choose>
    <Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />
</Project>
```

See also

- Conditional constructs
- Project file schema reference

Import element (MSBuild)

5/10/2019 • 2 minutes to read • Edit Online

Imports the contents of one project file into another project file.

<Project> <Import>

Syntax

```
<Import Project="ProjectPath"
    Condition="'String A'=='String B'" />
```

Attributes and elements

The following sections describe attributes, child elements, and parent elements.

Attributes

ATTRIBUTE	DESCRIPTION
`Project`	Required attribute. The path of the project file to import. The path can include wildcards. The matching files are imported in sorted order. By using this feature, you can add code to a project just by adding the code file to a directory.
`Condition`	Optional attribute. A condition to be evaluated. For more information, see Conditions.
`Sdk`	Optional attribute. References a project SDK.

Child elements

None

Parent elements

ELEMENT	DESCRIPTION
Project	Required root element of an MSBuild project file.
ImportGroup	Contains a collection of `Import` elements grouped under an optional condition.

Remarks

By using the `Import` element, you can reuse code that is common to many project files. This makes it easier to

maintain the code because any updates you make to the shared code get propagated to all the projects that import it.

By convention, shared imported project files are saved as *.targets* files, but they are standard MSBuild project files. MSBuild does not prevent you from importing a project that has a different file name extension, but we recommend that you use the *.targets* extension for consistency.

Relative paths in imported projects are interpreted relative to the directory of the importing project. Therefore, if a project file is imported into several project files in different locations, the relative paths in the imported project file will be interpreted differently for each imported project.

All MSBuild reserved properties that relate to the project file, for example, `MSBuildProjectDirectory` and `MSBuildProjectFile`, that are referenced in an imported project are assigned values based on the importing project file.

If the imported project does not have a `DefaultTargets` attribute, imported projects are inspected in the order that they are imported, and the value of the first discovered `DefaultTargets` attribute is used. For example, if ProjectA imports ProjectB and ProjectC (in that order), and ProjectB imports ProjectD, MSBuild first looks for `DefaultTargets` specified on ProjectA, then ProjectB, then ProjectD, and finally ProjectC.

The schema of an imported project is identical to that of a standard project. Although MSBuild may be able to build an imported project, it is unlikely because an imported project typically does not contain information about which properties to set or the order in which to run targets. The imported project depends on the project into which it is imported to provide that information.

Wildcards

In the .NET Framework 4, MSBuild allows wildcards in the Project attribute. When there are wildcards, all matches found are sorted (for reproducibility), and then they are imported in that order as if the order had been explicitly set.

This is useful if you want to offer an extensibility point so that someone else can import a file without requiring you to explicitly add the file name to the importing file. For this purpose, *Microsoft.Common.Targets* contains the following line at the top of the file.

```
<Import Project="$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\$(MSBuildThisFile)\ImportBefore\*"
Condition="'$(ImportByWildcardBeforeMicrosoftCommonTargets)' == 'true' and
exists('$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\$(MSBuildThisFile)\ImportBefore')"/>
```

Example

The following example shows a project that has several items and properties and imports a general project file.

```
<Project DefaultTargets="Compile"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <PropertyGroup>
        <resourcefile>Strings.resx</resourcefile>

        <compiledresources>
            $(O)\$(MSBuildProjectName).Strings.resources
        </compiledresources>
    </PropertyGroup>

    <ItemGroup>
        <CSFile Include="*.cs" />

        <Reference Include="System" />
        <Reference Include="System.Data" />
    </ItemGroup>

    <Import Project="$(CommonLocation)\General.targets" />
</Project>
```

See also

- Project file schema reference
- How to: Use the same target in multiple project files

ImportGroup element

2/21/2019 • 2 minutes to read • Edit Online

Contains a collection of `Import` elements that are grouped under an optional condition. For more information, see Import element (MSBuild).

<Project> <ImportGroup>

Syntax

```
<ImportGroup Condition="'String A' == 'String B'">
    <Import ... />
    <Import ... />
</ImportGroup>
```

Attributes and elements

The following sections describe attributes, child elements, and parent elements.

Attributes

ATTRIBUTE	DESCRIPTION
`Condition`	Optional attribute. The condition to be evaluated. For more information, see Conditions.

Child elements

ELEMENT	DESCRIPTION
Import	Imports the contents of one project file into another project file.

Parent elements

ELEMENT	DESCRIPTION
Project	Required root element of an MSBuild project file.

Example

The following code example shows the `ImportGroup` element.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <ImportGroup>
        <Import Project="$(Targets1.targets)" />
        <Import Project="$(Targets2.targets)" />
    </ImportGroup>
...
</Project>
```

See also

- Project file schema reference
- Items

Item element (MSBuild)

4/18/2019 • 3 minutes to read • Edit Online

Contains a user-defined item and its metadata. Every item that is used in a MSBuild project must be specified as a child of an `ItemGroup` element.

<Project> <ItemGroup> <Item>

Syntax

```
<Item Include="*.cs"
        Exclude="MyFile.cs"
        Remove="RemoveFile.cs"
        Condition="'String A'=='String B'" >
    <ItemMetadata1>...</ItemMetadata1>
    <ItemMetadata2>...</ItemMetadata2>
</Item>
```

Specify metadata as attributes

In MSBuild 15.1 or later, any metadata with a name that doesn't conflict with the current list of attributes can optionally be expressed as an attribute.

For example, to specify a list of NuGet packages, you would normally use something like the following syntax.

```
<ItemGroup>
  <PackageReference Include="Newtonsoft.Json">
    <Version>9.0.1-beta1<Version>
  </PackageReference>
</ItemGroup>
```

Now, however, you can pass the `Version` metadata as an attribute, such as in the following syntax:

```
<ItemGroup>
  <PackageReference Include="Newtonsoft.Json" Version="9.0.1-beta1" />
</ItemGroup>
```

Attributes and elements

The following sections describe attributes, child elements, and parent elements.

Attributes

ATTRIBUTE	DESCRIPTION
`Include`	Optional attribute. The file or wildcard to include in the list of items.

ATTRIBUTE	DESCRIPTION
`Exclude`	Optional attribute. The file or wildcard to exclude from the list of items.
`Condition`	Optional attribute. The condition to be evaluated. For more information, see Conditions.
`Remove`	Optional attribute. The file or wildcard to remove from the list of items.
`KeepDuplicates`	Optional attribute. Specifies whether an item should be added to the target group if it's an exact duplicate of an existing item. If the source and target item have the same `Include` value but different metadata, the item is added even if `KeepDuplicates` is set to `false`. For more information, see Items. This attribute is valid only if it's specified for an item in an `ItemGroup` that's in a `Target`.
`KeepMetadata`	Optional attribute. The metadata for the source items to add to the target items. Only the metadata whose names are specified in the semicolon-delimited list are transferred from a source item to a target item. For more information, see Items. This attribute is valid only if it's specified for an item in an `ItemGroup` that's in a `Target`.
`RemoveMetadata`	Optional attribute. The metadata for the source items to not transfer to the target items. All metadata is transferred from a source item to a target item except metadata whose names are contained in the semicolon-delimited list of names. For more information, see Items. This attribute is valid only if it's specified for an item in an `ItemGroup` that's in a `Target`.
`Update`	Optional attribute. (Available only for .NET Core projects in Visual Studio 2017 or later.) Enables you to modify metadata of a file that was included by using a glob. This attribute is valid only if it's specified for an item in an `ItemGroup` that is not in a `Target`.

Child elements

ELEMENT	DESCRIPTION
ItemMetadata	A user-defined item metadata key, which contains the item metadata value. There may be zero or more `ItemMetadata` elements in an item.

Parent elements

ELEMENT	DESCRIPTION
ItemGroup	Grouping element for items.

Remarks

`Item` elements define inputs into the build system, and are grouped into item collections based on their user-defined collection names. These item collections can be used as parameters for tasks, which use the individual items in the collections to perform the steps of the build process. For more information, see Items.

Using the notation @(<myType>) enables a collection of items of type <myType> to be expanded into a semicolon-delimited list of strings, and passed to a parameter. If the parameter is of type `string`, then the value of the parameter is the list of elements, separated by semicolons. If the parameter is an array of strings (`string[]`), then each element is inserted into the array based on the location of the semicolons. If the task parameter is of type ITaskItem `[]`, then the value is the contents of the item collection together with any metadata attached. To delimit each item by using a character other than a semicolon, use the syntax @(<myType>, '<separator>').

The MSBuild engine can evaluate wildcards such as `*` and `?` and recursive wildcards such as */**/*.cs*. For more information, see Items.

Examples

The following code example shows how to declare two items of type `CSFile`. The second declared item contains metadata that has `MyMetadata` set to `HelloWorld`.

```
<ItemGroup>
    <CSFile Include="engine.cs; form.cs" />
    <CSFile Include="main.cs" >
        <MyMetadata>HelloWorld</MyMetadata>
    </CSFile>
</ItemGroup>
```

The following code example shows how to use the `Update` attribute to modify the metadata in a file called *somefile.cs* that was included via a glob. (Available only for .NET Core projects in Visual Studio 2017 or later.)

```
<ItemGroup>
    <Compile Update="somefile.cs">  // or Update="*.designer.cs"
        <MetadataKey>MetadataValue</MetadataKey>
    </Compile>
</ItemGroup>
```

See also

- Items
- Common MSBuild project items

- MSBuild properties
- Project file schema reference

ItemDefinitionGroup element (MSBuild)

2/21/2019 • 2 minutes to read • Edit Online

The `ItemDefinitionGroup` element lets you define a set of Item Definitions, which are metadata values that are applied to all items in the project, by default. ItemDefinitionGroup supersedes the need to use the CreateItem task and the CreateProperty task. For more information, see Item definitions.

<Project> <ItemDefinitionGroup>

Syntax

```
<ItemDefinitionGroup Condition="'String A' == 'String B'">
    <Item1>... </Item1>
    <Item2>... </Item2>
</ItemDefinitionGroup>
```

Attributes and elements

The following sections describe attributes, child elements, and parent elements.

Attributes

ATTRIBUTE	DESCRIPTION
`Condition`	Optional attribute. Condition to be evaluated. For more information, see Conditions.

Child elements

ELEMENT	DESCRIPTION
Item	Defines the inputs for the build process. There may be zero or more `Item` elements in an `ItemDefinitionGroup`.

Parent elements

ELEMENT	DESCRIPTION
Project	Required root element of an MSBuild project file.

Example

The following code example defines two metadata items, m and n, in an ItemDefinitionGroup. In this example, the default metadata "m" is applied to Item "i" because metadata "m" is not explicitly defined by Item "i". However, default metadata "n" is not applied to Item "i" because metadata "n" is already defined by Item "i".

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <ItemDefinitionGroup>
        <i>
            <m>m1</m>
            <n>n1</n>
        </i>
    </ItemDefinitionGroup>
    <ItemGroup>
        <i Include="a">
            <o>o1</o>
            <n>n2</n>
        </i>
    </ItemGroup>
    ...
</Project>
```

See also

- Project file schema reference
- Items

ItemGroup element (MSBuild)

2/21/2019 • 2 minutes to read • Edit Online

Contains a set of user-defined Item elements. Every item used in a MSBuild project must be specified as a child of an `ItemGroup` element.

<Project> <ItemGroup>

Syntax

```
<ItemGroup Condition="'String A' == 'String B'">
    <Item1>... </Item1>
    <Item2>... </Item2>
</ItemGroup>
```

Attributes and elements

The following sections describe attributes, child elements, and parent elements.

Attributes

ATTRIBUTE	DESCRIPTION
`Condition`	Optional attribute. Condition to be evaluated. For more information, see Conditions.

Child elements

ELEMENT	DESCRIPTION
Item	Defines the inputs for the build process. There may be zero or more `Item` elements in an `ItemGroup`.

Parent elements

ELEMENT	DESCRIPTION
Project	Required root element of an MSBuild project file.
Target	Starting with .NET Framework 3.5, the `ItemGroup` element can appear inside a `Target` element. For more information, see Targets.

Example

The following code example shows the user-defined item collections `Res` and `CodeFiles` declared inside of an `ItemGroup` element. Each of the items in the `Res` item collection contains a user-defined child ItemMetadata element.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <ItemGroup>
        <Res Include = "Strings.fr.resources" >
            <Culture>fr</Culture>
        </Res>
        <Res Include = "Dialogs.fr.resources" >
            <Culture>fr</Culture>
        </Res>

        <CodeFiles Include="**\*.cs" Exclude="**\generated\*.cs" />
        <CodeFiles Include="..\..\Resources\Constants.cs" />
    </ItemGroup>
...
</Project>
```

See also

- Project file schema reference
- Items
- Common MSBuild project items

ItemMetadata element (MSBuild)

2/21/2019 • 2 minutes to read • Edit Online

Contains a user-defined item metadata key, which contains the item metadata value. An item may have any number of metadata key-value pairs.

<Project> <ItemGroup> <Item>

Syntax

```
<ItemMetadataName> Item Metadata value</ItemMetadataName>
```

Attributes and elements

The following sections describe attributes, child elements, and parent elements.

Attributes

ATTRIBUTE	DESCRIPTION
`Condition`	Optional attribute. Condition to be evaluated. For more information, see Conditions.

Child elements

None.

Parent elements

ELEMENT	DESCRIPTION
Item	A user-defined element that defines the inputs for the build process.

Text value

A text value is optional.

This text specifies the item metadata value, which can be either text or XML.

Example

The following code example shows how to add `Culture` metadata with the value `fr` to the item `CSFile`.

```
<ItemGroup>
    <CSFile Include="main.cs" >
        <Culture>fr</Culture>
    </CSFile>
</ItemGroup>
```

See also

- Project file schema reference
- Items

OnError element (MSBuild)

2/21/2019 • 2 minutes to read • Edit Online

Causes one or more targets to execute, if the `ContinueOnError` attribute is `false` for a failed task.

<Project> <Target> <OnError>

Syntax

```
<OnError ExecuteTargets="TargetName"
    Condition="'String A'=='String B'" />
```

Attributes and elements

The following sections describe attributes, child elements, and parent elements.

Attributes

ATTRIBUTE	DESCRIPTION
`Condition`	Optional attribute. Condition to be evaluated. For more information, see Conditions.
`ExecuteTargets`	Required attribute. The targets to execute if a task fails. Separate multiple targets with semicolons. Multiple targets are executed in the order specified.

Child elements

None.

Parent elements

ELEMENT	DESCRIPTION
Target	Container element for MSBuild tasks.

Remarks

MSBuild executes the `OnError` element if one of the `Target` element's tasks fails with the `ContinueOnError` attribute set to `ErrorAndStop` (or `false`). When the task fails, the targets specified in the `ExecuteTargets` attribute is executed. If there is more than one `OnError` element in the target, the `OnError` elements are executed sequentially when the task fails.

For information about the `ContinueOnError` attribute, see Task element (MSBuild). For information about targets, see Targets.

Example

The following code executes the `TaskOne` and `TaskTwo` tasks. If `TaskOne` fails, MSBuild evaluates the `OnError` element and executes the `OtherTarget` target.

```
<Target Name="ThisTarget">
    <TaskOne ContinueOnError="ErrorAndStop">
    </TaskOne>
    <TaskTwo>
    </TaskTwo>
    <OnError ExecuteTargets="OtherTarget" />
</Target>
```

See also

- Project file schema reference
- Targets

Otherwise element (MSBuild)

2/21/2019 • 2 minutes to read • Edit Online

Specifies the block of code to execute if and only if the conditions of all `When` elements evaluate to `false`.

<Project> <Choose> <When> <Choose> ... <Otherwise> <Choose> ...

Syntax

```
<Otherwise>
    <PropertyGroup>... </PropertyGroup>
    <ItemGroup>... </ItemGroup>
    <Choose>... </Choose>
</Otherwise>
```

Attributes and elements

The following sections describe attributes, child elements, and parent elements.

Attributes

None.

Child elements

ELEMENT	DESCRIPTION
Choose	Optional element. Evaluates child elements to select one section of code to execute. There may be zero or more `Choose` elements in an `Otherwise` element.
ItemGroup	Optional element. Contains a set of user-defined Item elements. There may be zero or more `ItemGroup` elements in an `Otherwise` element.
PropertyGroup	Optional element. Contains a set of user-defined Property elements. There may be zero or more `PropertyGroup` elements in an `Otherwise` element.

Parent elements

ELEMENT	DESCRIPTION
Choose	Evaluates child elements to select one section of code to execute.

Remarks

There may be only one `Otherwise` element in a `Choose` element, and it must be last element.

The `Choose`, `When`, and `Otherwise` elements are used together to provide a way to select one section of code to execute out of a number of possible alternatives. For more information, see Conditional constructs.

Example

The following project uses the `Choose` element to select which set of property values in the `When` elements to set. If the `Condition` attributes of both `When` elements evaluate to `false`, the property values in the `Otherwise` element are set.

```
<Project
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003" >
    <PropertyGroup>
        <Configuration Condition="'$(Configuration)' == ''">Debug</Configuration>
        <OutputType>Exe</OutputType>
        <RootNamespace>ConsoleApplication1</RootNamespace>
        <AssemblyName>ConsoleApplication1</AssemblyName>
        <WarningLevel>4</WarningLevel>
    </PropertyGroup>
    <Choose>
        <When Condition=" '$(Configuration)'=='debug' ">
            <PropertyGroup>
                <DebugSymbols>true</DebugSymbols>
                <DebugType>full</DebugType>
                <Optimize>false</Optimize>
                <OutputPath>.\bin\Debug\</OutputPath>
                <DefineConstants>DEBUG;TRACE</DefineConstants>
            </PropertyGroup>
            <ItemGroup>
                <Compile Include="UnitTesting\*.cs" />
                <Reference Include="NUnit.dll" />
            </ItemGroup>
        </When>
        <When Condition=" '$(Configuration)'=='retail' ">
            <PropertyGroup>
                <DebugSymbols>false</DebugSymbols>
                <Optimize>true</Optimize>
                <OutputPath>.\bin\Release\</OutputPath>
                <DefineConstants>TRACE</DefineConstants>
            </PropertyGroup>
        </When>
        <Otherwise>
            <PropertyGroup>
                <DebugSymbols>true</DebugSymbols>
                <Optimize>false</Optimize>
                <OutputPath>.\bin\$(Configuration)\</OutputPath>
                <DefineConstants>DEBUG;TRACE</DefineConstants>
            </PropertyGroup>
        </Otherwise>
        </Choose>
    <Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />
</Project>
```

See also

- Conditional constructs
- Project file schema reference

Output element (MSBuild)

2/21/2019 • 2 minutes to read • Edit Online

Stores task output values in items and properties.

<Project> <Target> <Task> <Output>

Syntax

```
<Output TaskParameter="Parameter"
    PropertyName="PropertyName"
    Condition = "'String A' == 'String B'" />
```

Attributes and elements

The following sections describe attributes, child elements, and parent elements.

Attributes

ATTRIBUTE	DESCRIPTION
`TaskParameter`	Required attribute. The name of the task's output parameter.
`PropertyName`	Either the `PropertyName` or `ItemName` attribute is required. The property that receives the task's output parameter value. Your project can then reference the property with the $(<PropertyName>) syntax. This property name can either be a new property name or a name that is already defined in the project. This attribute cannot be used if `ItemName` is also being used.
`ItemName`	Either the `PropertyName` or `ItemName` attribute is required. The item that receives the task's output parameter value. Your project can then reference the item with the @(<ItemName>) syntax. The item name can either be a new item name or a name that is already defined in the project. When the item name is an existing item, the output parameter values are added to the existing item. This attribute cannot be used if `PropertyName` is also being used.
`Condition`	Optional attribute. Condition to be evaluated. For more information, see Conditions.

Child elements

None.

Parent elements

ELEMENT	DESCRIPTION
Task	Creates and executes an instance of an MSBuild task.

Example

The following code example shows the `Csc` task being executed inside of a `Target` element. The items and properties passed to the task parameters are declared outside of the scope of this example. The value from the output parameter `OutputAssembly` is stored in the `FinalAssemblyName` item, and the value from the output parameter `BuildSucceeded` is stored in the `BuildWorked` property. For more information, see Tasks.

```
<Target Name="Compile" DependsOnTargets="Resources">
    <Csc  Sources="@(CSFile)"
            TargetType="library"
            Resources="@(CompiledResources)"
            EmitDebugInformation="$(includeDebugInformation)"
            References="@(Reference)"
            DebugType="$(debuggingType)"
            OutputAssembly="$(builtdir)\$(MSBuildProjectName).dll" >
        <Output TaskParameter="OutputAssembly"
                  ItemName="FinalAssemblyName" />
        <Output TaskParameter="BuildSucceeded"
                  PropertyName="BuildWorked" />
    </Csc>
</Target>
```

See also

- Project file schema reference
- Tasks

Parameter element

2/21/2019 • 2 minutes to read • Edit Online

Contains information about a specific parameter for a task that is generated by a `UsingTask` `TaskFactory`. The name of the element is the name of the parameter. For more information, see UsingTask element (MSBuild).

<Project> <UsingTask> <ParameterGroup> <Parameter>

Syntax

```
<ParameterGroup ParameterType="SystemType"
    Output="true/false"
    Required="true/false" />
```

Attributes and elements

The following sections describe attributes, child elements, and parent elements.

Attributes

ATTRIBUTE	DESCRIPTION
`ParameterType`	Optional attribute. The .NET type of the parameter, for example, `System.String`.
`Output`	Optional Boolean attribute. If `true`, this parameter is an output parameter for the task. By default, the value is `false`.
`Required`	Optional Boolean attribute. If `true`, this parameter is a required parameter for the task. By default, the value is `false`.

Child elements

None.

Parent elements

ELEMENT	DESCRIPTION
ParameterGroup	Contains an optional list of parameters that will be present on the task that is generated by a `UsingTask` `TaskFactory`.

Example

The following example shows how to use the `Parameter` element.

```
<UsingTask TaskName="MyTask" AssemblyName="My.Assembly" TaskFactory="MyTaskFactory">
       <ParameterGroup>
              <Parameter1 ParameterType="System.String" Required="False" Output="False"/>
              <Parameter2 ParameterType="System.Int" Required="True" Output="False"/>
             ...
</ParameterGroup>
       <TaskBody Evaluate="true">
      ... Task factory-specific data ...
       </TaskBody>
</UsingTask>
```

See also

- Tasks
- Task reference
- Project file schema reference

ParameterGroup element

2/21/2019 • 2 minutes to read • Edit Online

Contains an optional list of parameters that will be present on the task that is generated by a `UsingTask` `TaskFactory`. For more information, see UsingTask element (MSBuild).

<Project> <UsingTask> <ParameterGroup>

Syntax

```
<ParameterGroup />
```

Attributes and elements

The following sections describe attributes, child elements, and parent elements.

Attributes

None.

Child elements

ELEMENT	DESCRIPTION
Parameter	Contains information about a specific parameter for a task that is generated by a `UsingTask` `TaskFactory`. The name of the element is the name of the parameter.

Parent elements

ELEMENT	DESCRIPTION
UsingTask	Provides a way to register tasks in MSBuild. There may be zero or more `UsingTask` elements in a project.

Example

The following example shows how to use the `ParameterGroup` element.

```
<UsingTask TaskName="MyTask" AssemblyName="My.Assembly" TaskFactory="MyTaskFactory">
       <ParameterGroup>
              <Parameter1 ParameterType="System.String" Required="False" Output="False"/>
              <Parameter2 ParameterType="System.Int" Required="True" Output="False"/>
             ...
</ParameterGroup>
       <TaskBody Evaluate="true">
      ... Task factory-specific data ...
       </TaskBody>
</UsingTask>
```

See also

- Tasks
- Task reference
- Project file schema reference

Project element (MSBuild)

3/28/2019 • 3 minutes to read • Edit Online

Required root element of an MSBuild project file.

Syntax

```
<Project InitialTargets="TargetA;TargetB"
         DefaultTargets="TargetC;TargetD"
         TreatAsLocalProperty="PropertyA;PropertyB"
         ToolsVersion=<version number>
         Sdk="name[/version]"
         xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <Sdk... />
    <Choose>... </Choose>
    <PropertyGroup>... </PropertyGroup>
    <ItemGroup>... </ItemGroup>
    <Target>... </Target>
    <UsingTask.../>
    <ProjectExtensions>... </ProjectExtensions>
    <Import... />
</Project>
```

Attributes and elements

The following sections describe attributes, child elements, and parent elements.

Attributes

ATTRIBUTE	DESCRIPTION
`DefaultTargets`	Optional attribute. The default target or targets to be the entry point of the build if no target has been specified. Multiple targets are semi-colon (;) delimited. If no default target is specified in either the `DefaultTargets` attribute or the MSBuild command line, the engine executes the first target in the project file after the Import elements have been evaluated.
`InitialTargets`	Optional attribute. The initial target or targets to be run before the targets specified in the `DefaultTargets` attribute or on the command line. Multiple targets are semi-colon (`;`) delimited. If multiple imported files define `InitialTargets` , all targets mentioned will be run, in the order the imports are encountered.

ATTRIBUTE	DESCRIPTION
`Sdk`	Optional attribute. The SDK name and optional version to use to create implicit Import statements that are added to the .proj file. If no version is specified, MSBuild will attempt to resolve a default version. For example, `<Project Sdk="Microsoft.NET.Sdk" />` or `<Project Sdk="My.Custom.Sdk/1.0.0" />`.
`ToolsVersion`	Optional attribute. The version of the Toolset MSBuild uses to determine the values for $(MSBuildBinPath) and $(MSBuildToolsPath).
`TreatAsLocalProperty`	Optional attribute. Property names that won't be considered to be global. This attribute prevents specific command-line properties from overriding property values that are set in a project or targets file and all subsequent imports. Multiple properties are semi-colon (;) delimited. Normally, global properties override property values that are set in the project or targets file. If the property is listed in the `TreatAsLocalProperty` value, the global property value doesn't override property values that are set in that file and any subsequent imports. For more information, see How to: Build the same source files with different options. **Note:** You set global properties at a command prompt by using the **-property** (or **-p**) switch. You can also set or modify global properties for child projects in a multi-project build by using the `Properties` attribute of the MSBuild task. For more information, see MSBuild task.
`xmlns`	Optional attribute. When specified, the `xmlns` attribute must have the value of `http://schemas.microsoft.com/developer/msbuild/2003`.

Child elements

ELEMENT	DESCRIPTION
Choose	Optional element. Evaluates child elements to select one set of `ItemGroup` elements and/or `PropertyGroup` elements to evaluate.
Import	Optional element. Enables a project file to import another project file. There may be zero or more `Import` elements in a project.

ELEMENT	DESCRIPTION
ImportGroup	Optional element. Contains a collection of `Import` elements that are grouped under an optional condition.
ItemGroup	Optional element. A grouping element for individual items. Items are specified by using the Item element. There may be zero or more `ItemGroup` elements in a project.
ItemDefinitionGroup	Optional element. Lets you define a set of Item Definitions, which are metadata values that are applied to all items in the project, by default. ItemDefinitionGroup supersedes the need to use the `CreateItem` task and the `CreateProperty` task.
ProjectExtensions	Optional element. Provides a way to persist non-MSBuild information in an MSBuild project file. There may be zero or one `ProjectExtensions` elements in a project.
PropertyGroup	Optional element. A grouping element for individual properties. Properties are specified by using the Property element. There may be zero or more `PropertyGroup` elements in a project.
Sdk	Optional element. References an MSBuild project SDK. This element can be used as an alternative to the Sdk attribute.
Target	Optional element. Contains a set of tasks for MSBuild to sequentially execute. Tasks are specified by using the Task element. There may be zero or more `Target` elements in a project.
UsingTask	Optional element. Provides a way to register tasks in MSBuild. There may be zero or more `UsingTask` elements in a project.

Parent elements

None.

See also

- How to: Specify which target to build first
- Command-line reference
- Project file schema reference
- MSBuild

ProjectExtensions element (MSBuild)

2/21/2019 • 2 minutes to read • Edit Online

Allows MSBuild project files to contain non-MSBuild information. Anything inside of a `ProjectExtensions` element will be ignored by MSBuild.

<Project> <ProjectExtensions>

Syntax

```
<ProjectExtensions>
    Non-MSBuild information to include in file.
</ProjectExtensions>
```

Attributes and elements

The following sections describe attributes, child elements, and parent elements.

Attributes

None

Child elements

None

Parent elements

ELEMENT	DESCRIPTION
Project	Required root element of an MSBuild project file.

Remarks

Only one `ProjectExtensions` element may be used in an MSBuild project.

Example

The following code example shows information from the integrated development environment being stored in a `ProjectExtensions` element.

```
<ProjectExtensions>
    <VSIDE>
        <External>
            <!--
            Raw XML passed to the IDE by an external source
            -->
        </External>
    </VSIDE>
</ProjectExtensions>
```

See also

- Project file schema reference
- MSBuild

Property element (MSBuild)

2/21/2019 • 2 minutes to read • Edit Online

Contains a user defined property name and value. Every property used in an MSBuild project must be specified as a child of a `PropertyGroup` element.

<Project> <PropertyGroup>

Syntax

```
<Property Condition="'String A' == 'String B'">
    Property Value
</Property>
```

Attributes and elements

The following sections describe attributes, child elements, and parent elements.

Attributes

ATTRIBUTE	DESCRIPTION
`Condition`	Optional attribute. Condition to be evaluated. For more information, see Conditions.

Child elements

None.

Parent elements

ELEMENT	DESCRIPTION
PropertyGroup	Grouping element for properties.

Text value

A text value is optional.

This text specifies the property value and may contain XML.

Remarks

Property names are limited to ASCII chars only. Property values are referenced in the project by placing the property name between "`$(`" and "`)`". For example, `$(builddir)\classes` would resolve to *build\classes*, if the `builddir` property had the value `build`. For more information on properties, see MSBuild properties.

Example

The following code sets the `Optimization` property to `false` and the `DefaultVersion` property to `1.0` if the

`Version` property is empty.

```
<PropertyGroup>
    <Optimization>false</Optimization>
    <DefaultVersion Condition="'$(Version)' == '" >1.0</DefaultVersion>
</PropertyGroup>
```

See also

- MSBuild properties
- Project file schema reference

PropertyGroup element (MSBuild)

2/21/2019 • 2 minutes to read • Edit Online

Contains a set of user-defined Property elements. Every `Property` element used in an MSBuild project must be a child of a `PropertyGroup` element.

<Project> <PropertyGroup>

Syntax

```
<PropertyGroup Condition="'String A' == 'String B'">
    <Property1>...</Property1>
    <Property2>...</Property2>
</PropertyGroup>
```

Attributes and elements

The following sections describe attributes, child elements, and parent elements.

Attributes

ATTRIBUTE	DESCRIPTION
Condition	Optional attribute. Condition to be evaluated. For more information, see Conditions.

Child elements

ELEMENT	DESCRIPTION
Property	Optional element. A user defined property name, which contains the property value. There may be zero or more *Property* elements in a `PropertyGroup` element.

Parent elements

ELEMENT	DESCRIPTION
Project	Required root element of an MSBuild project file.

Example

The following code example shows how to set properties based on a condition. In this example, if the value of the `CompileConfig` property is `DEBUG`, the `Optimization`, `Obfuscate`, and `OutputPath` properties inside of the `PropertyGroup` element are set.

```
<PropertyGroup Condition="'$(CompileConfig)' == 'DEBUG'" >
    <Optimization>false</Optimization>
    <Obfuscate>false</Obfuscate>
    <OutputPath>$(OutputPath)\debug</OutputPath>
</PropertyGroup>
```

See also

- Project file schema reference
- MSBuild properties

Sdk element (MSBuild)

4/16/2019 • 2 minutes to read • Edit Online

References an MSBuild project SDK.

<Project> <Sdk>

Syntax

```
<Sdk Name="My.Custom.Sdk"
     Version="1.0.0" />
```

Attributes and elements

The following sections describe attributes, child elements, and parent elements.

Attributes

ATTRIBUTE	DESCRIPTION
`Name`	Required attribute. The name of the project SDK.
`Version`	Optional attribute. The version of the project SDK

Child elements

None.

Parent elements

ELEMENT	DESCRIPTION
Project	Required root element of an MSBuild project file.

See also

- How to: Reference an MSBuild project SDK
- Project file schema reference
- MSBuild

Target element (MSBuild)

11/26/2019 • 4 minutes to read • Edit Online

Contains a set of tasks for MSBuild to execute sequentially.

<Project> <Target>

Syntax

```
<Target Name="Target Name"
        Inputs="Inputs"
        Outputs="Outputs"
        Returns="Returns"
        KeepDuplicateOutputs="true/false"
        BeforeTargets="Targets"
        AfterTargets="Targets"
        DependsOnTargets="DependentTarget"
        Condition="'String A' == 'String B'"
        Label="Label">
    <Task>... </Task>
    <PropertyGroup>... </PropertyGroup>
    <ItemGroup>... </ItemGroup>
    <OnError... />
</Target>
```

Attributes and elements

The following sections describe attributes, child elements, and parent elements.

Attributes

ATTRIBUTE	DESCRIPTION
`Name`	Required attribute. The name of the target.
`Condition`	Optional attribute. The condition to be evaluated. If the condition evaluates to `false`, the target will not execute the body of the target or any targets that are set in the `DependsOnTargets` attribute. For more information about conditions, see Conditions.
`Inputs`	Optional attribute. The files that form inputs into this target. Multiple files are separated by semicolons. The timestamps of the files will be compared with the timestamps of files in `Outputs` to determine whether the `Target` is up to date. For more information, see Incremental builds, How to: Build incrementally, and Transforms.

ATTRIBUTE	DESCRIPTION
`Outputs`	Optional attribute. The files that form outputs into this target. Multiple files are separated by semicolons. The timestamps of the files will be compared with the timestamps of files in `Inputs` to determine whether the `Target` is up to date. For more information, see Incremental builds, How to: Build incrementally, and Transforms.
`Returns`	Optional attribute. The set of items that will be made available to tasks that invoke this target, for example, MSBuild tasks. Multiple targets are separated by semicolons. If the targets in the file have no `Returns` attributes, the Outputs attributes are used instead for this purpose.
`KeepDuplicateOutputs`	Optional Boolean attribute. If `true`, multiple references to the same item in the target's Returns are recorded. By default, this attribute is `false`.
`BeforeTargets`	Optional attribute. A semicolon-separated list of target names. When specified, indicates that this target should run before the specified target or targets. This lets the project author extend an existing set of targets without modifying them directly. For more information, see Target build order.
`AfterTargets`	Optional attribute. A semicolon-separated list of target names. When specified, indicates that this target should run after the specified target or targets. This lets the project author extend an existing set of targets without modifying them directly. For more information, see Target build order.
`DependsOnTargets`	Optional attribute. The targets that must be executed before this target can be executed or top-level dependency analysis can occur. Multiple targets are separated by semicolons.
`Label`	Optional attribute. An identifier that can identify or order system and user elements.

Child elements

ELEMENT	DESCRIPTION
Task	Creates and executes an instance of an MSBuild task. There may be zero or more tasks in a target.

ELEMENT	DESCRIPTION
PropertyGroup	Contains a set of user-defined `Property` elements. Starting in the .NET Framework 3.5, a `Target` element may contain `PropertyGroup` elements.
ItemGroup	Contains a set of user-defined `Item` elements. Starting in the .NET Framework 3.5, a `Target` element may contain `ItemGroup` elements. For more information, see Items.
OnError	Causes one or more targets to execute if the `ContinueOnError` attribute is ErrorAndStop (or `false`) for a failed task. There may be zero or more `OnError` elements in a target. If `OnError` elements are present, they must be the last elements in the `Target` element. For information about the `ContinueOnError` attribute, see Task element (MSBuild).

Parent elements

ELEMENT	DESCRIPTION
Project	Required root element of an MSBuild project file.

Remarks

The first target to execute is specified at run time. Targets can have dependencies on other targets. For example, a target for deployment depends on a target for compilation. The MSBuild engine executes dependencies in the order in which they appear in the `DependsOnTargets` attribute, from left to right. For more information, see Targets.

MSBuild is import-order dependent, and the last definition of a target with a specific `Name` attribute is the definition used.

A target is only executed once during a build, even if more than one target has a dependency on it.

If a target is skipped because its `Condition` attribute evaluates to `false`, it can still be executed if it is invoked later in the build and its `Condition` attribute evaluates to `true` at that time.

Before MSBuild 4, `Target` returned any items that were specified in the `Outputs` attribute. To do this, MSBuild had to record these items in case tasks later in the build requested them. Because there was no way to indicate which targets had outputs that callers would require, MSBuild accumulated all items from all `Outputs` on all invoked `Target`s. This lead to scaling problems for builds that had a large number of output items.

If the user specifies a `Returns` on any `Target` element in a project, then only those `Target`s that have a `Returns` attribute record those items.

A `Target` may contain both an `Outputs` attribute and a `Returns` attribute. `Outputs` is used with `Inputs` to determine whether the target is up-to-date. `Returns`, if present, overrides the value of `Outputs` to determine which items are returned to callers. If `Returns` is not present, then `Outputs` will be made available to callers except in the case described earlier.

Before MSBuild 4, any time that a `Target` included multiple references to the same item in its `Outputs`, those duplicate items would be recorded. In very large builds that had a large number of outputs and many project interdependencies, this would cause a large amount of memory to be wasted because the duplicate items were

not of any use. When the `KeepDuplicateOutputs` attribute is set to `true`, these duplicates are recorded.

Example

The following code example shows a `Target` element that executes the `Csc` task.

```
<Target Name="Compile" DependsOnTargets="Resources" Returns="$(TargetPath)">
    <Csc Sources="@(CSFile)"
          TargetType="library"
          Resources="@(CompiledResources)"
          EmitDebugInformation="$(includeDebugInformation)"
          References="@(Reference)"
          DebugType="$(debuggingType)" >
        <Output TaskParameter="OutputAssembly"
                 ItemName="FinalAssemblyName" />
    </Csc>
</Target>
```

See also

- Targets
- Project file schema reference

Task element (MSBuild)

2/21/2019 • 2 minutes to read • Edit Online

Creates and executes an instance of an MSBuild task. The element name is determined by the name of the task being created.

<Project> <Target>

Syntax

```
<Task Parameter1="Value1"... ParameterN="ValueN"
    ContinueOnError="WarnAndContinue/true/ErrorAndContinue/ErrorAndStop/false"
    Condition="'String A' == 'String B'" >
    <Output... />
</Task>
```

Attributes and elements

The following sections describe attributes, child elements, and parent elements.

Attributes

ATTRIBUTE	DESCRIPTION
`Condition`	Optional attribute. Condition to be evaluated. For more information, see Conditions.
`ContinueOnError`	Optional attribute. Can contain one of the following values: - **WarnAndContinue** or **true**. When a task fails, subsequent tasks in the Target element and the build continue to execute, and all errors from the task are treated as warnings. - **ErrorAndContinue**. When a task fails, subsequent tasks in the `Target` element and the build continue to execute, and all errors from the task are treated as errors. - **ErrorAndStop** or **false** (default). When a task fails, the remaining tasks in the `Target` element and the build aren't executed, and the entire `Target` element and the build is considered to have failed. Versions of the .NET Framework before 4.5 supported only the `true` and `false` values. For more information, see How to: Ignore errors in tasks.
`Parameter`	Required if the task class contains one or more properties labeled with the `[Required]` attribute. A user-defined task parameter that contains the parameter value as its value. There can be any number of parameters in the `Task` element, with each attribute mapping to a .NET property in the task class.

Child elements

ELEMENT	DESCRIPTION
Output	Stores outputs from the task in the project file. There may be zero or more `Output` elements in a task.

Parent elements

ELEMENT	DESCRIPTION
Target	Container element for MSBuild tasks.

Remarks

A `Task` element in an MSBuild project file creates an instance of a task, sets properties on it, and executes it. The `Output` element stores output parameters in properties or items to be used elsewhere in the project file.

If there are any OnError elements in the parent `Target` element of a task, they will still be evaluated if the task fails and `ContinueOnError` has a value of `false`. For more information on tasks, see Tasks.

Example

The following code example creates an instance of the Csc task class, sets six of the properties, and executes the task. After execution, the value of the `OutputAssembly` property of the object is placed into an item list named `FinalAssemblyName`.

```
<Target Name="Compile" DependsOnTarget="Resources" >
    <Csc Sources="@(CSFile)"
          TargetType="library"
          Resources="@(CompiledResources)"
          EmitDebugInformation="$(includeDebugInformation)"
          References="@(Reference)"
          DebugType="$(debuggingType)" >
        <Output TaskParameter="OutputAssembly"
                ItemName="FinalAssemblyName" />
    </Csc>
</Target>
```

See also

- Tasks
- Task reference
- Project file schema reference

TaskBody element (MSBuild)

2/21/2019 • 2 minutes to read • Edit Online

Contains the data that is passed to a `UsingTask` `TaskFactory`. For more information, see UsingTask element (MSBuild).

<Project> <UsingTask> <TaskBody>

Syntax

```
<TaskBody Evaluate="true/false" />
```

Attributes and elements

The following sections describe attributes, child elements, and parent elements.

Attributes

ATTRIBUTE	DESCRIPTION
`Evaluate`	Optional Boolean attribute. If `true`, MSBuild evaluates any inner elements, and expands items and properties before it passes the information to the `TaskFactory` when the task is instantiated.

Child elements

ELEMENT	DESCRIPTION
Data	The text between the `TaskBody` tags is sent verbatim to the `TaskFactory`.

Parent elements

ELEMENT	DESCRIPTION
UsingTask	Provides a way to register tasks in MSBuild. There may be zero or more `UsingTask` elements in a project.

Example

The following example shows how to use the `TaskBody` element with an `Evaluate` attribute.

```
<UsingTask TaskName="MyTask" AssemblyName="My.Assembly" TaskFactory="MyTaskFactory">
       <ParameterGroup>
              <Parameter1 ParameterType="System.String" Required="False" Output="False"/>
              <Parameter2 ParameterType="System.Int" Required="True" Output="False"/>
              ...
</ParameterGroup>
       <TaskBody Evaluate="true">
      ... Task factory-specific data ...
       </TaskBody>
</UsingTask>
```

See also

- Tasks
- Task reference
- Project file schema reference

UsingTask element (MSBuild)

10/4/2019 • 2 minutes to read • Edit Online

Maps the task that is referenced in a Task element to the assembly that contains the task implementation.

<Project> <UsingTask>

Syntax

```
<UsingTask TaskName="TaskName"
    AssemblyName = "AssemblyName"
    TaskFactory = "ClassName"
    Condition="'String A'=='String B'" />
```

NOTE

Unlike properties and items, the *first* `UsingTask` element that applies to a `TaskName` will be used; to override tasks you must define a new `UsingTask` *before* the existing one.

Attributes and elements

The following sections describe attributes, child elements, and parent elements.

Attributes

ATTRIBUTE	DESCRIPTION
`AssemblyName`	Either the `AssemblyName` attribute or the `AssemblyFile` attribute is required. The name of the assembly to load. The `AssemblyName` attribute accepts strong-named assemblies, although strong-naming is not required. Using this attribute is equivalent to loading an assembly by using the Load method in .NET. You cannot use this attribute if the `AssemblyFile` attribute is used.
`AssemblyFile`	Either the `AssemblyName` or the `AssemblyFile` attribute is required. The file path of the assembly. This attribute accepts full paths or relative paths. Relative paths are relative to the directory of the project file or targets file where the `UsingTask` element is declared. Using this attribute is equivalent to loading an assembly by using the LoadFrom method in .NET. You cannot use this attribute if the `AssemblyName` attribute is used.

ATTRIBUTE	DESCRIPTION
`TaskFactory`	Optional attribute. Specifies the class in the assembly that is responsible for generating instances of the specified `Task` name. The user may also specify a `TaskBody` as a child element that the task factory receives and uses to generate the task. The contents of the `TaskBody` are specific to the task factory.
`TaskName`	Required attribute. The name of the task to reference from an assembly. If ambiguities are possible, this attribute should always specify full namespaces. If there are ambiguities, MSBuild chooses an arbitrary match, which could produce unexpected results.
`Condition`	Optional attribute. The condition to evaluate. For more information, see Conditions.

Child elements

ELEMENT	DESCRIPTION
ParameterGroup	The set of parameters that appear on the task that is generated by the specified `TaskFactory`.
Task	The data that is passed to the `TaskFactory` to generate an instance of the task.

Parent elements

ELEMENT	DESCRIPTION
Project	Required root element of an MSBuild project file.

Remarks

Environment variables, command-line properties, project-level properties, and project-level items can be referenced in the `UsingTask` elements included in the project file either directly or through an imported project file. For more information, see Tasks.

> **NOTE**
> Project-level properties and items have no meaning if the `UsingTask` element is coming from one of the *.tasks* files that are globally registered with the MSBuild engine. Project-level values are not global to MSBuild.

In MSBuild 4.0, using tasks can be loaded from *.overridetask* files.

Example

The following example shows how to use the `UsingTask` element with an `AssemblyName` attribute.

```
<UsingTask TaskName="MyTask" AssemblyName="My.Assembly" TaskFactory="MyTaskFactory">
       <ParameterGroup>
              <Parameter1 ParameterType="System.String" Required="False" Output="False"/>
              <Parameter2 ParameterType="System.Int" Required="True" Output="False"/>
              ...
</ParameterGroup>
       <TaskBody>
      ... Task factory-specific data ...
       </TaskBody>
</UsingTask>
```

Example

The following example shows how to use the `UsingTask` element with an `AssemblyFile` attribute.

```
<UsingTask TaskName="Email"
              AssemblyFile="c:\myTasks\myTask.dll" />
```

See also

- Tasks
- Task reference
- Project file schema reference

When element (MSBuild)

2/21/2019 • 2 minutes to read • Edit Online

Specifies a possible block of code for the `Choose` element to select.

<Project> <Choose> <When> <Choose> ... <Otherwise> <Choose> ...

Syntax

```
<When Condition="'StringA'=='StringB'">
    <PropertyGroup>... </PropertyGroup>
    <ItemGroup>... </ItemGroup>
    <Choose>... </Choose>
</When>
```

Attributes and elements

The following sections describe attributes, child elements, and parent elements.

Attributes

ATTRIBUTE	DESCRIPTION
Condition	Required attribute. Condition to evaluate. For more information, see Conditions.

Child elements

ELEMENT	DESCRIPTION
Choose	Optional element. Evaluates child elements to select one section of code to execute. There may be zero or more `Choose` elements in a `When` element.
ItemGroup	Optional element. Contains a set of user-defined Item elements. There may be zero or more `ItemGroup` elements in a `When` element.
PropertyGroup	Optional element. Contains a set of user-defined Property elements. There may be zero or more `PropertyGroup` elements in an `When` element.

Parent elements

ELEMENT	DESCRIPTION
Choose element (MSBuild)	Evaluates child elements to select one section of code to execute.

Remarks

If the `Condition` attribute evaluates to true, the child `ItemGroup` and `PropertyGroup` elements of the `When` element are executed and all subsequent `When` elements are skipped.

The `Choose`, `When`, and `Otherwise` elements are used together to provide a way to select one section of code to execute out of a number of possible alternatives. For more information, see Conditional constructs.

Example

The following project uses the `Choose` element to select which set of property values in the `When` elements to set. If the `Condition` attributes of both `When` elements evaluate to `false`, the property values in the `Otherwise` element are set.

```
<Project
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003" >
    <PropertyGroup>
        <Configuration Condition="'$(Configuration)' == ''">Debug</Configuration>
        <OutputType>Exe</OutputType>
        <RootNamespace>ConsoleApplication1</RootNamespace>
        <AssemblyName>ConsoleApplication1</AssemblyName>
        <WarningLevel>4</WarningLevel>
    </PropertyGroup>
    <Choose>
        <When Condition=" '$(Configuration)'=='debug' ">
            <PropertyGroup>
                <DebugSymbols>true</DebugSymbols>
                <DebugType>full</DebugType>
                <Optimize>false</Optimize>
                <OutputPath>.\bin\Debug\</OutputPath>
                <DefineConstants>DEBUG;TRACE</DefineConstants>
            </PropertyGroup>
            <ItemGroup>
                <Compile Include="UnitTesting\*.cs" />
                <Reference Include="NUnit.dll" />
            </ItemGroup>
        </When>
        <When Condition=" '$(Configuration)'=='retail' ">
            <PropertyGroup>
                <DebugSymbols>false</DebugSymbols>
                <Optimize>true</Optimize>
                <OutputPath>.\bin\Release\</OutputPath>
                <DefineConstants>TRACE</DefineConstants>
            </PropertyGroup>
        </When>
        <Otherwise>
            <PropertyGroup>
                <DebugSymbols>true</DebugSymbols>
                <Optimize>false</Optimize>
                <OutputPath>.\bin\$(Configuration)\</OutputPath>
                <DefineConstants>DEBUG;TRACE</DefineConstants>
            </PropertyGroup>
        </Otherwise>
    </Choose>
    <Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />
</Project>
```

See also

- Conditional constructs
- Project file schema reference

MSBuild task reference

10/21/2019 • 5 minutes to read • Edit Online

Tasks provide the code that runs during the build process. The tasks in the following list are included with MSBuild. When Visual C++ is installed, additional tasks are available that are used to build Visual C++ projects. For more information, see C++ tasks.

In addition to the parameters listed in the topics in this section, each task also has the following parameters:

PARAMETER	DESCRIPTION
`Condition`	Optional `String` parameter. A `Boolean` expression that the MSBuild engine uses to determine whether this task will be executed. For information about the conditions that are supported by MSBuild, see Conditions.
`ContinueOnError`	Optional parameter. Can contain one of the following values: - **WarnAndContinue** or **true**. When a task fails, subsequent tasks in the Target element and the build continue to execute, and all errors from the task are treated as warnings. - **ErrorAndContinue**. When a task fails, subsequent tasks in the `Target` element and the build continue to execute, and all errors from the task are treated as errors. - **ErrorAndStop** or **false** (default). When a task fails, the remaining tasks in the `Target` element and the build aren't executed, and the entire `Target` element and the build is considered to have failed. Versions of the .NET Framework before 4.5 supported only the `true` and `false` values. For more information, see How to: Ignore errors in tasks.

In this section

- Task base class

Adds several parameters to the tasks that derive from the Task class.

- TaskExtension base class

Adds several parameters to the tasks that derive from the TaskExtension class.

- ToolTaskExtension base class

Adds several parameters to the tasks that derive from the ToolTaskExtension class.

- AL (Assembly Linker) task

Creates an assembly with a manifest from one or more files that are either modules or resource files.

- AspNetCompiler task

Wraps *aspnet_compiler.exe*, a utility to precompile ASP.NET applications.

- AssignCulture task

Assigns culture identifiers to items.

- AssignProjectConfiguration task

Accepts a list of configuration strings and assigns them to specified projects.

- AssignTargetPath task

Accepts a list of files and adds `<TargetPath>` attributes if they are not already specified.

- CallTarget task

Invokes a target in the project file.

- CombinePath task

Combines the specified paths into a single path.

- ConvertToAbsolutePath task

Converts a relative path or reference into an absolute path.

- Copy task

Copies files to a new location.

- CreateCSharpManifestResourceName task

Creates a Visual C#-style manifest name from a given *.resx* file name or other resource.

- CreateItem task

Populates item collections from the input items, allowing items to be copied from one list to another.

- CreateProperty task

Populates properties from the input values, allowing values to be copied from one property or string to another.

- CreateVisualBasicManifestResourceName task

Creates a Visual Basic-style manifest name from a given *.resx* file name or other resource.

- Csc task

Invokes the Visual C# compiler to produce executables, dynamic-link libraries, or code modules.

- Delete task

Deletes the specified files.

- DownloadFile task

Downloads a file to the specified location.

- Error task

Stops a build and logs an error based on an evaluated conditional statement.

- Exec task

Runs the specified program or command with the specified arguments.

- FindAppConfigFile task

Finds the *app.config* file, if any, in the provided lists.

- FindInList task

Finds an item in a specified list that has the matching itemspec.

- FindUnderPath task

Determines which items in the specified item collection exist in the specified folder and all of its subfolders.

- FormatUrl task

Converts a URL to a correct URL format.

- FormatVersion task

Appends the revision number to the version number.

- GenerateApplicationManifest task

Generates a ClickOnce application manifest or a native manifest.

- GenerateBootstrapper task

Provides an automated way to detect, download, and install an application and its prerequisites.

- GenerateDeploymentManifest task

Generates a ClickOnce deployment manifest.

- GenerateResource task

Converts *.txt* and *.resx* files to common language runtime binary *.resources* files.

- GenerateTrustInfo task

Generates the application trust from the base manifest, and from the `TargetZone` and `ExcludedPermissions` parameters.

- GetAssemblyIdentity task

Retrieves the assembly identities from the specified files and outputs the identity information.

- GetFileHash task

Computes checksums of the contents of a file or set of files.

- GetFrameworkPath task

Retrieves the path to the .NET Framework assemblies.

- GetFrameworkSdkPath task

Retrieves the path to the Windows Software Development Kit (SDK).

- GetReferenceAssemblyPaths task

Returns the reference assembly paths of the various frameworks.

- LC task

Generates a *.license* file from a *.licx* file.

- MakeDir task

Creates directories and, if necessary, any parent directories.

- Message task

Logs a message during a build.

- Move task

Moves files to a new location.

- MSBuild task

Builds MSBuild projects from another MSBuild project.

- ReadLinesFromFile task

Reads a list of items from a text file.

- RegisterAssembly task

Reads the metadata within the specified assembly and adds the necessary entries to the registry.

- RemoveDir task

Removes the specified directories and all of its files and subdirectories.

- RemoveDuplicates task

Removes duplicate items from the specified item collection.

- RequiresFramework35SP1Assembly task

Determines whether the application requires the .NET Framework 3.5 SP1.

- ResGen Task

Obsolete. Use the GenerateResource task task to convert *.txt* and *.resx* files to and from common language runtime binary *.resources* files.

- ResolveAssemblyReference task

Determines all assemblies that depend on the specified assemblies.

- ResolveComReference task

Takes a list of one or more type library names or *.tlb* files and resolves those type libraries to locations on disk.

- ResolveKeySource task

Determines the strong name key source

- ResolveManifestFiles task

Resolves the following items in the build process to files for manifest generation: built items, dependencies, satellites, content, debug symbols, and documentation.

- ResolveNativeReference task

Resolves native references.

- ResolveNonMSBuildProjectOutput task

Determines the output files for non-MSBuild project references.

- SGen task

Creates an XML serialization assembly for types in the specified assembly.

- SignFile task

Signs the specified file using the specified certificate.

- Touch task

Sets the access and modification times of files.

- UnregisterAssembly task

Unregisters the specified assemblies for COM interop purposes.

- Unzip task

Unzips a *.zip* archive to the specified location.

- UpdateManifest task

Updates selected properties in a manifest and resigns.

- Vbc task

Invokes the Visual Basic compiler to produce executables, dynamic-link libraries, or code modules..

- VerifyFileHash task

Verifies that a file matches the expected file hash.

- Warning task

Logs a warning during a build based on an evaluated conditional statement.

- WriteCodeFragment task

Generates a temporary code file by using the specified generated code fragment. Does not delete the file.

- WriteLinesToFile task

Writes the specified items to the specified text file.

- XmlPeek task

Returns values as specified by XPath query from an XML file.

- XmlPoke task

Sets values as specified by an XPath query into an XML file.

- XslTransformation task

Transforms an XML input by using an *Extensible Stylesheet Language Transformation* (XSLT) or compiled XSLT and outputs to an output device or a file.

- ZipDirectory task

Creates a *.zip* archive from the contents of a directory.

See also

- MSBuild reference
- Task writing
- Tasks

MSBuild tasks specific to C++

10/21/2019 • 2 minutes to read • Edit Online

Tasks provide the code that runs during the build process. When C++ is installed, the following tasks are available, in addition to those that are installed with MSBuild. For more information, see MSBuild (C++) overview.

In addition to the parameters for each task, every task also has the following parameters.

PARAMETER	DESCRIPTION
`Condition`	Optional `String` parameter. A `Boolean` expression that the MSBuild engine uses to determine whether this task will be executed. For information about the conditions that are supported by MSBuild, see Conditions.
`ContinueOnError`	Optional parameter. Can contain one of the following values: - **WarnAndContinue** or **true**. When a task fails, subsequent tasks in the Target element and the build continue to execute, and all errors from the task are treated as warnings - **ErrorAndContinue**. When a task fails, subsequent tasks in the `Target` element and the build continue to execute, and all errors from the task are treated as errors. - **ErrorAndStop** or **false** (default). When a task fails, the remaining tasks in the `Target` element and the build aren't executed, and the entire `Target` element and the build are considered to have failed. Versions of the .NET Framework before 4.5 supported only the `true` and `false` values. For more information, see How to: Ignore errors in tasks.

Related topics

TITLE	DESCRIPTION
BscMake task	Wraps the Microsoft Browse Information Maintenance Utility tool (*bscmake.exe*).
CL task	Wraps the C++ compiler tool (*cl.exe*).
CPPClean task	Deletes the temporary files that MSBuild creates when a C++ project is built.
ClangCompile task	Wraps the C++ compiler tool (*clang.exe*).
CustomBuild task	Wraps the C++ compiler tool (*cmd.exe*).
FXC task	Use HLSL shader compilers in the build process.

TITLE	DESCRIPTION
GetOutOfDateItems	Reads old tlogs, writes new tlogs and returns set of items which are not up-to-date. (helper task)
GetOutputFileName	Gets output file name for cl and other tools, which allow specifying only output directory or full file name or nothing. (helper task)
LIB task	Wraps the Microsoft 32-Bit Library Manager tool (*lib.exe*).
Link task	Wraps the C++ linker tool (*link.exe*).
MIDL task	Wraps the Microsoft Interface Definition Language (MIDL) compiler tool (*midl.exe*).
MT task	Wraps the Microsoft Manifest Tool (*mt.exe*).
MultiToolTask task	No description.
ParallelCustomBuild task	Run parallel instances of the CustomBuild task.
RC task	Wraps the Microsoft Windows Resource Compiler tool (*rc.exe*).
SetEnv task	Sets or deletes the value of a specified environment variable.
TrackedVCToolTask base class	Inherits from VCToolTask.
VCMessage task	Logs warning messages and error messages during a build. (Not extendable. Internal use only.)
VCToolTask base class	Inherits from ToolTask.
XDCMake task	Wraps the XML Documentation tool (*xdcmake.exe*), which merges XML document comment (*.xdc*) files into an *.xml* file.
XSD task	Wraps the XML Schema Definition tool (*xsd.exe*), which generates schema or class files from a source. *See note below.*
MSBuild reference	Describes the elements of the MSBuild system.
Tasks	Describes tasks, which are units of code that can be combined to produce a build.
Task writing	Describes how to create a task.

NOTE

Starting in Visual Studio 2017, C++ project support for *xsd.exe* is deprecated. You can still use the **Microsoft.VisualC.CppCodeProvider** APIs by manually adding *CppCodeProvider.dll* to the GAC.

BscMake task

10/21/2019 • 2 minutes to read • Edit Online

> **IMPORTANT**
> BscMake is no longer used by the Visual Studio IDE. Since Visual Studio 2008, browse information is stored automatically in an *.sdf* file in the *Solution* folder.

Wraps the Microsoft Browse Information Maintenance Utility tool (*bscmake.exe*). The *bscmake.exe* tool builds a browse information file (*.bsc*) from source browser files (*.sbr*) that are created during compilation. Use the **Object Browser** to view a *.bsc* file. For more information, see BSCMAKE reference.

Parameters

The following table describes the parameters of the **BscMake** task. Most task parameters correspond to a command-line option.

PARAMETER	DESCRIPTION
AdditionalOptions	Optional **String** parameter. A list of options as specified on the command line. For example, /<option1> /<option2> /<option#>. Use this parameter to specify options that are not represented by any other **BscMake** task parameter. For more information, see the options in BSCMAKE options.
OutputFile	Optional **String** parameter. Specifies a file name that overrides the default output file name. For more information, see the **/o** option in BSCMAKE options.
PreserveSBR	Optional **Boolean** parameter. If `true`, forces a nonincremental build. A full, nonincremental build occurs regardless of whether a *.bsc* file exists, and prevents *.sbr* files from being truncated. For more information, see the **/n** option in BSCMAKE options.
Sources	Optional **ITaskItem[]** parameter. Defines an array of MSBuild source file items that can be consumed and emitted by tasks.

PARAMETER	DESCRIPTION
SuppressStartupBanner	Optional **Boolean** parameter. If `true`, prevents the display of the copyright and version number message when the task starts. For more information, see the **/NOLOGO** option in BSCMAKE options.
TrackerLogDirectory	Optional **String** parameter. Specifies the directory for the tracker log.

See also

- Task reference

CL task

10/21/2019 • 13 minutes to read • Edit Online

Wraps the Microsoft C++ compiler tool, *cl.exe*. The compiler produces executable (*.exe*) files, dynamic-link library (*.dll*) files, or code module (*.netmodule*) files. For more information, see Compiler options.

Parameters

The following list describes the parameters of the **CL** task. Most task parameters, and a few sets of parameters, correspond to a command-line option.

- **AdditionalIncludeDirectories**

  Optional String[] parameter.

  Adds a directory to the list of directories that are searched for include files.

  For more information, see /I (Additional include directories).

- **AdditionalOptions**

  Optional String parameter.

  A list of command-line options. For example, "/<option1> /<option2> /<option#>". Use this parameter to specify command-line options that are not represented by any other task parameter.

  For more information, see Compiler options.

- **AdditionalUsingDirectories**

  Optional String[] parameter.

  Specifies a directory that the compiler will search to resolve file references passed to the **#using** directive.

  For more information, see /AI (Specify metadata directories).

- **AlwaysAppend**

  Optional String parameter.

  A string that always gets emitted on the command line. Its default value is "**/c**".

- **AssemblerListingLocation**

  Creates a listing file that contains assembly code.

  For more information, see the **/Fa** option in /FA, /Fa (Listing file).

- **AssemblerOutput**

  Optional String parameter.

  Creates a listing file that contains assembly code.

  Specify one of the following values, each of which corresponds to a command-line option.

  - **NoListing** - *<none>*
  - **AssemblyCode** - **/FA**

- **AssemblyAndMachineCode** - **/FAc**
- **AssemblyAndSourceCode** - **/FAs**
- **All** - **/FAcs**

  For more information, see the **/FA**, **/FAc**, **/FAs**, and **/FAcs** options in /FA, /Fa (Listing file).

- **BasicRuntimeChecks**

  Optional String parameter.

  Enables and disables the run-time error checks feature, in conjunction with the runtime_checks pragma.

  Specify one of the following values, each of which corresponds to a command-line option.

  - **Default** - *<none>*
  - **StackFrameRuntimeCheck** - **/RTCs**
  - **UninitializedLocalUsageCheck** - **/RTCu**
  - **EnableFastChecks** - **/RTC1**

    For more information, see /RTC (Run-time error checks).

- **BrowseInformation**

  Optional Boolean parameter.

  If `true`, creates a browse information file.

  For more information, see the **/FR** option in /FR, /Fr (Create .sbr file).

- **BrowseInformationFile**

  Optional String parameter.

  Specifies a file name for the browse information file.

  For more information, see the **BrowseInformation** parameter in this table, and also see /FR, /Fr (Create .sbr file).

- **BufferSecurityCheck**

  Optional Boolean parameter.

  If `true`, detects some buffer overruns that overwrite the return address, a common technique for exploiting code that does not enforce buffer size restrictions.

  For more information, see /GS (Buffer security check).

- **BuildingInIDE**

  Optional Boolean parameter.

  If `true`, indicates that **MSBuild** is invoked by the IDE. Otherwise, **MSBuild** is invoked on the command line.

- **CallingConvention**

  Optional String parameter.

  Specifies the calling convention, which determines the order in which function arguments are pushed onto the stack, whether the caller function or called function removes the arguments from the stack at the end of

the call, and the name-decorating convention that the compiler uses to identify individual functions.

Specify one of the following values, each of which corresponds to a command-line option.

- **Cdecl** - **/Gd**
- **FastCall** - **/Gr**
- **StdCall** - **/Gz**

  For more information, see /Gd, /Gr, /Gv, /Gz (Calling convention).

- **CompileAs**

  Optional String parameter.

  Specifies whether to compile the input file as a C or C++ source file.

  Specify one of the following values, each of which corresponds to a command-line option.

  - **Default** - *<none>*
  - **CompileAsC** - **/TC**
  - **CompileAsCpp** - **/TP**

    For more information, see /Tc, /Tp, /TC, /TP (Specify source file type).

- **CompileAsManaged**

  Optional String parameter.

  Enables applications and components to use features from the common language runtime (CLR).

  Specify one of the following values, each of which corresponds to a command-line option.

  - **false** - *<none>*
  - **true** - **/clr**
  - **Pure** - **/clr:pure**
  - **Safe** - **/clr:safe**
  - **OldSyntax** - **/clr:oldSyntax**

    For more information, see /clr (Common language runtime compilation).

- **CreateHotpatchableImage**

  Optional Boolean parameter.

  If `true`, tells the compiler to prepare an image for *hot patching*. This parameter ensures that the first instruction of each function is two bytes, which is required for hot patching.

  For more information, see /hotpatch (Create hotpatchable image).

- **DebugInformationFormat**

  Optional String parameter.

  Selects the type of debugging information created for your program and whether this information is kept in object (*.obj*) files or in a program database (PDB).

  Specify one of the following values, each of which corresponds to a command-line option.

- **OldStyle** - **/Z7**
- **ProgramDatabase** - **/Zi**
- **EditAndContinue** - **/ZI**

  For more information, see /Z7, /Zi, /ZI (Debug information format).

- **DisableLanguageExtensions**

  Optional Boolean parameter.

  If **true**, tells the compiler to emit an error for language constructs that are not compatible with either ANSI C or ANSI C++.

  For more information, see the **/Za** option in /Za, /Ze (Disable language extensions).

- **DisableSpecificWarnings**

  Optional String[] parameter.

  Disables the warning numbers that are specified in a semicolon-delimited list.

  For more information, see the `/wd` option in /w, /W0, /W1, /W2, /W3, /W4, /w1, /w2, /w3, /w4, /Wall, /wd, /we, /wo, /Wv, /WX (Warning level).

- **EnableEnhancedInstructionSet**

  Optional String parameter.

  Specifies the architecture for code generation that uses the Streaming SIMD Extensions (SSE) and Streaming SIMD Extensions 2 (SSE2) instructions.

  Specify one of the following values, each of which corresponds to a command-line option.

  - **StreamingSIMDExtensions** - **/arch:SSE**
  - **StreamingSIMDExtensions2** - **/arch:SSE2**

    For more information, see /arch (x86).

- **EnableFiberSafeOptimizations**

  Optional Boolean parameter.

  If `true`, support fiber safety for data allocated by using static thread-local storage, that is, data allocated by using `__declspec(thread)`.

  For more information, see /GT (Support fiber-safe thread-local storage).

- **EnablePREfast**

  Optional Boolean parameter.

  If `true`, enable code analysis.

  For more information, see /analyze (Code analysis).

- **ErrorReporting**

  Optional String parameter.

  Lets you provide internal compiler error (ICE) information directly to Microsoft. By default, the setting in IDE builds is **Prompt** and the setting in command-line builds is **Queue**.

Specify one of the following values, each of which corresponds to a command-line option.

- **None** - **/errorReport:none**
- **Prompt** - **/errorReport:prompt**
- **Queue** - **/errorReport:queue**
- **Send** - **/errorReport:send**

  For more information, see /errorReport (Report internal compiler errors).

- **ExceptionHandling**

  Optional String parameter.

  Specifies the model of exception handling to be used by the compiler.

  Specify one of the following values, each of which corresponds to a command-line option.

  - **false** - *<none>*
  - **Async** - **/EHa**
  - **Sync** - **/EHsc**
  - **SyncCThrow** - **/EHs**

    For more information, see /EH (Exception handling model).

- **ExpandAttributedSource**

  Optional Boolean parameter.

  If `true`, creates a listing file that has expanded attributes injected into the source file.

  For more information, see /Fx (Merge injected code).

- **FavorSizeOrSpeed**

  Optional String parameter.

  Specifies whether to favor code size or code speed.

  Specify one of the following values, each of which corresponds to a command-line option.

  - **Neither** - *<none>*
  - **Size** - **/Os**
  - **Speed** - **/Ot**

    For more information, see /Os, /Ot (Favor small code, favor fast code).

- **FloatingPointExceptions**

  Optional Boolean parameter.

  If `true`, enables the reliable floating-point exception model. Exceptions will be raised immediately after they are triggered.

  For more information, see the **/fp:except** option in /fp (Specify floating-point behavior).

- **FloatingPointModel**

  Optional String parameter.

Sets the floating point model.

Specify one of the following values, each of which corresponds to a command-line option.

- **Precise** - **/fp:precise**
- **Strict** - **/fp:strict**
- **Fast** - **/fp:fast**

  For more information, see /fp (Specify floating-point behavior).

- **ForceConformanceInForLoopScope**

  Optional Boolean parameter.

  If `true`, implements standard C++ behavior in for loops that use Microsoft extensions (/Ze).

  For more information, see /Zc:forScope (Force conformance in for loop scope).

- **ForcedIncludeFiles**

  Optional `String[]` parameter.

  Causes the preprocessor to process one or more specified header files.

  For more information, see /FI (Name forced include file).

- **ForcedUsingFiles**

  Optional **String[]** parameter.

  Causes the preprocessor to process one or more specified **#using** files.

  For more information, see /FU (Name forced #using file).

- **FunctionLevelLinking**

  Optional `Boolean` parameter.

  If `true`, enables the compiler to package individual functions in the form of packaged functions (COMDATs).

  For more information, see /Gy (Enable function-level linking).

- **GenerateXMLDocumentationFiles**

  Optional `Boolean` parameter.

  If `true`, causes the compiler to process documentation comments in source code files and to create an *.xdc* file for each source code file that has documentation comments.

  For more information, see /doc (Process documentation comments) (C/C++). Also see the **XMLDocumentationFileName** parameter in this table.

- **IgnoreStandardIncludePath**

  Optional `Boolean` parameter.

  If `true`, prevents the compiler from searching for include files in directories specified in the PATH and INCLUDE environment variables.

  For more information, see /X (Ignore standard include paths).

- **InlineFunctionExpansion**

Optional **String** parameter.

Specifies the level of inline function expansion for the build.

Specify one of the following values, each of which corresponds to a command-line option.

- **Default** - *<none>*
- **Disabled** - **/Ob0**
- **OnlyExplicitInline** - **/Ob1**
- **AnySuitable** - **/Ob2**

  For more information, see /Ob (Inline function expansion).

- **IntrinsicFunctions**

  Optional `Boolean` parameter.

  If `true`, replaces some function calls with intrinsic or otherwise special forms of the function that help your application run faster.

  For more information, see /Oi (Generate intrinsic functions).

- **MinimalRebuild**

  Optional `Boolean` parameter.

  If `true`, enables minimal rebuild, which determines whether C++ source files that include changed C++ class definitions (stored in header (.h) files) must be recompiled.

  For more information, see /Gm (Enable minimal rebuild).

- **MultiProcessorCompilation**

  Optional `Boolean` parameter.

  If `true`, use multiple processors to compile. This parameter creates a process for each effective processor on your computer.

  For more information, see /MP (Build with multiple processes). Also, see the **ProcessorNumber** parameter in this table.

- **ObjectFileName**

  Optional **String** parameter.

  Specifies an object (.obj) file name or directory to be used instead of the default.

  For more information, see /Fo (Object file name).

- **ObjectFiles**

  Optional **String[]** parameter.

  A list of object files.

- **OmitDefaultLibName**

  Optional `Boolean` parameter.

  If `true`, omits the default C run-time library name from the object (*.obj*) file. By default, the compiler puts the name of the library into the *.obj* file to direct the linker to the correct library.

For more information, see /Zl (Omit default library name).

- **OmitFramePointers**

  Optional `Boolean` parameter.

  If `true`, suppresses creation of frame pointers on the call stack.

  For more information, see /Oy (Frame-pointer omission).

- **OpenMPSupport**

  Optional `Boolean` parameter.

  If `true`, causes the compiler to process OpenMP clauses and directives.

  For more information, see /openmp (Enable OpenMP 2.0 support).

- **Optimization**

  Optional **String** parameter.

  Specifies various code optimizations for speed and size.

  Specify one of the following values, each of which corresponds to a command-line option.

  - **Disabled** - **/Od**
  - **MinSpace** - **/O1**
  - **MaxSpeed** - **/O2**
  - **Full** - **/Ox**

    For more information, see /O Options (Optimize code).

- **PrecompiledHeader**

  Optional **String** parameter.

  Create or use a precompiled header (*.pch*) file during the build.

  Specify one of the following values, each of which corresponds to a command-line option.

  - **NotUsing** - *<none>*
  - **Create** - **/Yc**
  - **Use** - **/Yu**

    For more information, see /Yc (Create precompiled header file) and /Yu (Use precompiled header file). Also, see the **PrecompiledHeaderFile** and **PrecompiledHeaderOutputFile** parameters in this table.

- **PrecompiledHeaderFile**

  Optional **String** parameter.

  Specifies a precompiled header file name to create or use.

  For more information, see /Yc (Create precompiled header file) and /Yu (Use precompiled header file).

- **PrecompiledHeaderOutputFile**

  Optional **String** parameter.

Specifies a path name for a precompiled header instead of using the default path name.

For more information, see /Fp (Name .pch file).

- **PreprocessKeepComments**

  Optional `Boolean` parameter.

  If `true`, preserves comments during preprocessing.

  For more information, see /C (Preserve comments during preprocessing).

- **PreprocessorDefinitions**

  Optional `String[]` parameter.

  Defines a preprocessing symbol for your source file.

  For more information, see /D (Preprocessor definitions).

- **PreprocessOutput**

  Optional `ITaskItem[]` parameter.

  Defines an array of preprocessor output items that can be consumed and emitted by tasks.

- **PreprocessOutputPath**

  Optional `String` parameter.

  Specifies the name of the output file to which the **PreprocessToFile** parameter writes preprocessed output.

  For more information, see /Fi (Preprocess output file name).

- **PreprocessSuppressLineNumbers**

  Optional `Boolean` parameter.

  If `true`, preprocesses C and C++ source files and copies the preprocessed files to the standard output device.

  For more information, see /EP (Preprocess to stdout without #line directives).

- **PreprocessToFile**

  Optional `Boolean` parameter.

  If `true`, preprocesses C and C++ source files and writes the preprocessed output to a file.

  For more information, see /P (Preprocess to a file).

- **ProcessorNumber**

  Optional `Integer` parameter.

  Specifies the maximum number of processors to use in a multiprocessor compilation. Use this parameter in combination with the **MultiProcessorCompilation** parameter.

- **ProgramDataBaseFileName**

  Optional `String` parameter.

  Specifies a file name for the program database (PDB) file.

  For more information, see /Fd (Program database file name).

- **RuntimeLibrary**

  Optional `String` parameter.

  Indicates whether a multithreaded module is a DLL, and selects retail or debug versions of the run-time library.

  Specify one of the following values, each of which corresponds to a command-line option.

  - **MultiThreaded** - **/MT**
  - **MultiThreadedDebug** - **/MTd**
  - **MultiThreadedDLL** - **/MD**
  - **MultiThreadedDebugDLL** - **/MDd**

    For more information, see /MD, /MT, /LD (Use run-time library).

- **RuntimeTypeInfo**

  Optional `Boolean` parameter.

  If `true`, adds code to check C++ object types at run time (run-time type information).

  For more information, see /GR (Enable run-time type information).

- **ShowIncludes**

  Optional `Boolean` parameter.

  If `true`, causes the compiler to output a list of the include files.

  For more information, see /showIncludes (List include files).

- **SmallerTypeCheck**

  Optional `Boolean` parameter.

  If `true`, reports a run-time error if a value is assigned to a smaller data type and causes a data loss.

  For more information, see the **/RTCc** option in /RTC (Run-time error checks).

- **Sources**

  Required `ITaskItem[]` parameter.

  Specifies a list of source files separated by spaces.

- **StringPooling**

  Optional `Boolean` parameter.

  If `true`, enables the compiler to create one copy of identical strings in the program image.

  For more information, see /GF (Eliminate duplicate strings).

- **StructMemberAlignment**

  Optional `String` parameter.

  Specifies the byte alignment for all members in a structure.

  Specify one of the following values, each of which corresponds to a command-line option.

  - **Default** - **/Zp1**

- **1Byte** - **/Zp1**
- **2Bytes** - **/Zp2**
- **4Bytes** - **/Zp4**
- **8Bytes** - **/Zp8**
- **16Bytes** - **/Zp16**

  For more information, see /Zp (Struct member alignment).

- **SuppressStartupBanner**

  Optional `Boolean` parameter.

  If `true`, prevents the display of the copyright and version number message when the task starts.

  For more information, see /nologo (Suppress startup banner) (C/C++).

- **TrackerLogDirectory**

  Optional `String` parameter.

  Specifies the intermediate directory where tracking logs for this task are stored.

  For more information, see the **TLogReadFiles** and **TLogWriteFiles** parameters in this table.

- **TreatSpecificWarningsAsErrors**

  Optional **String[]** parameter.

  Treats the specified list of compiler warnings as errors.

  For more information, see the **/we** `n` option in /w, /W0, /W1, /W2, /W3, /W4, /w1, /w2, /w3, /w4, /Wall, /wd, /we, /wo, /Wv, /WX (Warning level).

- **TreatWarningAsError**

  Optional `Boolean` parameter.

  If `true`, treat all compiler warnings as errors.

  For more information, see **/WX** option in /w, /W0, /W1, /W2, /W3, /W4, /w1, /w2, /w3, /w4, /Wall, /wd, /we, /wo, /Wv, /WX (Warning level).

- **TreatWChar_tAsBuiltInType**

  Optional `Boolean` parameter.

  If `true`, treat the `wchar_t` type as a native type.

  For more information, see /Zc:wchar_t (wchar_t is native type).

- **UndefineAllPreprocessorDefinitions**

  Optional `Boolean` parameter.

  If `true`, undefines the Microsoft-specific symbols that the compiler defines.

  For more information, see the **/u** option in /U, /u (Undefine symbols).

- **UndefinePreprocessorDefinitions**

Optional `String[]` parameter.

Specifies a list of one or more preprocessor symbols to undefine.

For more information, see **/U** option in /U, /u (Undefine symbols).

- **UseFullPaths**

  Optional `Boolean` parameter.

  If `true`, displays the full path of source code files passed to the compiler in diagnostics.

  For more information, see /FC (Full path of source code file in diagnostics).

- **UseUnicodeForAssemblerListing**

  Optional `Boolean` parameter.

  If `true`, causes the output file to be created in UTF-8 format.

  For more information, see the **/FAu** option in /FA, /Fa (Listing file).

- **WarningLevel**

  Optional `String` parameter.

  Specifies the highest level of warning that is to be generated by the compiler.

  Specify one of the following values, each of which corresponds to a command-line option.

  - **TurnOffAllWarnings** - **/W0**
  - **Level1** - **/W1**
  - **Level2** - **/W2**
  - **Level3** - **/W3**
  - **Level4** - **/W4**
  - **EnableAllWarnings** - **/Wall**

    For more information, see the **/W***n* option in /w, /W0, /W1, /W2, /W3, /W4, /w1, /w2, /w3, /w4, /Wall, /wd, /we, /wo, /Wv, /WX (Warning level).

- **WholeProgramOptimization**

  Optional `Boolean` parameter.

  If `true`, enables whole program optimization.

  For more information, see /GL (Whole program optimization).

- **XMLDocumentationFileName**

  Optional `String` parameter.

  Specifies the name of the generated XML documentation files. This parameter can be a file or directory name.

  For more information, see the `name` argument in /doc (Process documentation comments) (C/C++). Also see the **GenerateXMLDocumentationFiles** parameter in this table.

- **MinimalRebuildFromTracking**

Optional `Boolean` parameter.

If `true`, a tracked incremental build is performed; if `false`, a rebuild is performed.

- **TLogReadFiles**

  Optional `ITaskItem[]` parameter.

  Specifies an array of items that represent the *read file tracking logs*.

  A read-file tracking log (*.tlog*) contains the names of the input files that are read by a task, and is used by the project build system to support incremental builds. For more information, see the **TrackerLogDirectory** and **TrackFileAccess** parameters in this table.

- **TLogWriteFiles**

  Optional `ITaskItem[]` parameter.

  Specifies an array of items that represent the *write file tracking logs*.

  A write-file tracking log (*.tlog*) contains the names of the output files that are written by a task, and is used by the project build system to support incremental builds. For more information, see the **TrackerLogDirectory** and **TrackFileAccess** parameters in this table.

- **TrackFileAccess**

  Optional `Boolean` parameter.

  If `true`, tracks file access patterns.

  For more information, see the **TLogReadFiles** and **TLogWriteFiles** parameters in this table.

See also

- Task reference

CPPClean Task

10/21/2019 • 2 minutes to read • Edit Online

Deletes the temporary files that MSBuild creates when a C++ project is built. The process of deleting build files is known as *cleaning*.

Parameters

The following table describes the parameters of the **CPPClean** task.

PARAMETER	DESCRIPTION
DeletedFiles	Optional `ITaskItem[]` output parameter. Defines an array of MSBuild output file items that can be consumed and emitted by tasks.
DoDelete	Optional **Boolean** parameter. If `true`, clean temporary build files.
FilePatternsToDeleteOnClean	Required `String` parameter. Specifies a semicolon-delimited list of file extensions of files to clean.
FilesExcludedFromClean	Optional `String` parameter. Specifies a semicolon-delimited list of files not to clean.
FoldersToClean	Required `String` parameter. Specifies a semicolon-delimited list of directories to clean. You can specify a full or a relative path, and the path can contain the wildcard symbol (*).

See also

- Task reference

ClangCompile task

10/21/2019 • 4 minutes to read • Edit Online

Wraps the Microsoft C++ compiler tool, clang.exe.

Parameters

The following table describes the parameters of the **ClangCompile** task.

PARAMETER	DESCRIPTION
AdditionalIncludeDirectories	Optional **string[]** parameter. Specifies one or more directories to add to the include path; separate with semi-colons if more than one. Use `-I[path]`.
AdditionalOptions	Optional **string** parameter.
BufferSecurityCheck	Optional **string** parameter. The Security Check helps detect stack-buffer over-runs, a common attempted attack upon a program's security. Use `fstack-protector`.
BuildingInIde	Optional **bool** parameter.
CLanguageStandard	Optional **string** parameter. Determines the C language standard. Use `std=[value]` with value of **c89**, **c99**, **c11**, **gnu99**, or **gnu11**.
ClangVersion	Optional **string** parameter.
CompileAs	Optional **string** parameter. Select compile language option for .c and .cpp files. Default will detect based on .c or .cpp extention. Use `-x c`, `-x c++`.
CppLanguageStandard	Optional **string** parameter. Determines the C++ language standard. Use `std=[value]` with value of **c++98**, **c++11**, **c++1y**, **gnu++98**, **gnu++11**, or **gnu++1y**.

PARAMETER	DESCRIPTION
DataLevelLinking	Optional **bool** parameter. Enables linker optimizations to remove unused data by emitting each data item in a separate section.
DebugInformationFormat	Optional **string** parameter. Specifies the type of debugging information generated by the compiler. **None**, produces no debugging information, so compilation may be faster (use `g0`). **FullDebug**, generate DWARF2 debug information (use `g2 -gdwarf-2`). **LineNumber**, generate Line Number information only (use `gline-tables-only`).
EnableNeonCodegen	Optional **bool** parameter. Enables code generation for NEON floating point hardware. This is applicable for arm architecture only.
ExceptionHandling	Optional **string** parameter. Specifies the model of exception handling to be used by the compiler. **Disabled**, disable exception handling (use `fno-exceptions`). **Enabled**, enable exception handling (use `fexceptions`). **UnwindTables**, generates any needed static data, but does not affect the code generated (use `funwind-tables`).
FloatABI	Optional **string** parameter. Selection option to choose the floating point ABI. **soft**, causes compiler to generate output containing library calls for floating-point operations (use `mfloat-abi=soft`). **softfp**, allows the generation of code using hardware floating-point instructions, but still uses the soft-float calling conventions (use `mfloat-abi=softfp`). **hard**, allows generation of floating-point instructions and uses FPU-specific calling conventions (use `mfloat-abi=hard`).
ForcedIncludeFiles	Optional **string[]** parameter. One or more forced include files. Use `-include [name]`.
FunctionLevelLinking	Optional **bool** parameter. Allows the compiler to package individual functions in the form of packaged functions (COMDATs). Required for edit and continue to work. Use `ffunction-sections`.

PARAMETER	DESCRIPTION
GccToolChain	Optional **string** parameter. Folder path to Gcc Tool Chain.
GNUMode	Optional **bool** parameter.
MSCompatibility	Optional **bool** parameter. Enable full Microsoft C++ compatibility.
MSCompatibilityVersion	Optional **string** parameter. Dot-separated value representing the Microsoft compiler version number to report in _MSC_VER (0 = don't define it (default)).
MSExtensions	Optional **bool** parameter. Accept some non-standard constructs supported by the Microsoft compiler.
MSCompilerVersion	Optional **string** parameter. Microsoft compiler version number to report in _MSC_VER (0 = don't define it (default)).
MSVCErrorReport	Optional **bool** parameter. Report errors which Visual Studio can use to parse for file and line information.
ObjectFileName	Optional **string** parameter. Specifies a name to override the default object file name; can be file or directory name. Use `/Fo[name]`.
OmitFramePointers	Optional **bool** parameter. Suppresses creation of frame pointers on the call stack.
Optimization	Optional **string** parameter. Specifies the optimization level for the application. **Custom**, custom optimization. **Disabled**, disable optimization (use `O0`). **MinSize**, optimize for size (use `Os`). **MaxSpeed**, optimize for speed (use `O2`). **Full**, expensive optimizations (use `O3`).

PARAMETER	DESCRIPTION
PositionIndependentCode	Optional **bool** parameter. Generate Position Independent Code (PIC) for use in a shared library.
PrecompiledHeader	Optional **string** parameter. Enables creation or use of a precompiled header during the build.
PrecompiledHeaderFile	Optional **string** parameter. Specifies header file name to use for precompiled header file. This file will be also added to **Forced Include Files** during build.
PrecompiledHeaderOutputFileDirectory	Optional **string** parameter. Specifies the directory for the generated precompiled header. This directory will be also added to **Additional Include Directories** during build.
PrecompiledHeaderCompileAs	Optional **string** parameter. Select compile language option for precompiled header file. Use `-x c-header`, `-x c++-header`.
PreprocessorDefinitions	Optional **string[]** parameter. Defines a preprocessing symbols for your source file. Use `-D`.
RuntimeLibrary	Optional **string** parameter. Specify runtime library for linking. Use `MSVC /MT`, `/MTd`, `/MD`, `/MDd` switches. **MultiThreaded**, causes your application to use the multithread, static version of the run-time library. **MultiThreadedDebug**, defines _DEBUG and _MT. This option also causes the compiler to place the library name LIBCMTD.lib into the .obj file so that the linker will use LIBCMTD.lib to resolve external symbols. **MultiThreadedDLL**, causes your application to use the multithread- and DLL-specific version of the run-time library. Defines _MT and _DLL and causes the compiler to place the library name MSVCRT.lib into the .obj file. **MultiThreadedDebugDLL**, defines _DEBUG, _MT, and _DLL and causes your application to use the debug multithread- and DLL-specific version of the run-time library. It also causes the compiler to place the library name MSVCRTD.lib into the .obj file.

PARAMETER	DESCRIPTION
RuntimeTypeInfo	Optional **bool** parameter. Adds code for checking C++ object types at run time (runtime type information). Use `frtti`, `fno-rtti`.
ShowIncludes	Optional **bool** parameter. Generates a list of include files with compiler output. Use `-H`.
Sources	Required **ITaskItem[]** parameter.
StrictAliasing	Optional **bool** parameter. Assume the strictest aliasing rules. An object of one type will never be assumed to reside at the same address as an object of a different type.
Sysroot	Optional **string** parameter. Folder path to the root directory for headers and libraries.
TargetArch	Optional **string** parameter. Target Architecture.
ThumbMode	Optional **string** parameter. Generate code that executes for thumb microarchitecture. This is applicable for arm architecture only. **Thumb**, generate Thumb code (use `mthumb`). **ARM**, generate Arm code (use `marm`). **Disabled**, option not applicable for chosen platform.
TrackerLogDirectory	Optional **string** parameter. Tracker Log Directory.
TreatWarningAsError	Optional **bool** parameter. Treats all compiler warnings as errors. For a new project, it may be best to use `/WX` in all compilations; resolving all warnings will ensure the fewest possible hard-to-find code defects.
UndefinePreprocessorDefinitions	Optional **string[]** parameter. Specifies one or more preprocessor undefines. Use `-U [macro]`.

PARAMETER	DESCRIPTION
UndefineAllPreprocessorDefinitions	Optional **bool** parameter. Undefine all previously defined preprocessor values. Use `-undef`.
UseMultiToolTask	Optional **bool** parameter. Multi-processor Compilation.
UseShortEnums	Optional **bool** parameter. Enum type uses only as many bytes required by input set of possible values.
Verbose	Optional **bool** parameter. Show commands to run and use verbose output.
WarningLevel	Optional **string** parameter. Select how strict you want the compiler to be about code errors. Other flags should be added directly to **Additional Options** (se `/w`, `/Weverything`). **TurnOffAllWarnings**, disables all compiler warnings (use `w`). **EnableAllWarnings**, enables all warnings, including those disabled by default (use `Wall`).

See also

Task reference

CustomBuild task

10/21/2019 • 2 minutes to read • Edit Online

Wraps the Microsoft C++ compiler tool, cmd.exe. This class derives from TrackedVCToolTask, but does not use file tracking to discover file dependencies. All dependencies should be explicitly specified as AdditionalDependencies for incremental build working properly.

Parameters

The following table describes the parameters of the **CustomBuild** task.

PARAMETER	DESCRIPTION
BuildSuffix	Optional **string** parameter.
Sources	Required **ITaskItem[]** parameter.
TrackerLogDirectory	Optional **string** parameter.

See also

Task reference

FXC task

10/21/2019 • 2 minutes to read • Edit Online

Use HLSL shader compilers in the build process.

Parameters

The following table describes the parameters of the **FXC** task.

PARAMETER	DESCRIPTION
AdditionalIncludeDirectories	Optional **string[]** parameter. Specifies one or more directories to add to the include path; separate with semi-colons if more than one. Use `/I[path]`.
AdditionalOptions	Optional **string** parameter.
AllResourcesBound	Optional **bool** parameter. Compiler will assume that all resources that a shader may reference are bound and are in good state for the duration of shader execution. Available for Shader Model 5.1 and above. Use `/all_resources_bound`.
AssemblerOutput	Optional **string** parameter. Specifies the contents of assembly language output file. Use `/Fc, /Fx`. **NoListing** **AssemblyCode**, use `Fc`. **AssemblyCodeAndHex**, use `Fx`.
AssemblerOutputFile	Optional **string** parameter. Specifies file name for assembly code listing file.
CompileD2DCustomEffect	Optional **bool** parameter. Compile a Direct2D custom effect that contains pixel shaders. Do not use for a vertex or compute custom effect.
ConsumeExportFile	Optional **string** parameter.

PARAMETER	DESCRIPTION
DisableOptimizations	Optional **bool** parameter. Disable optimizations. `/Od` implies `/Gfp` though output may not be identical to `/Od /Gfp`.
EnableDebuggingInformation	Optional **bool** parameter. Enable debugging information.
EnableUnboundedDescriptorTables	Optional **bool** parameter. Inform the compiler that a shader may contain a declaration of a resource array with unbounded range. Available for Shader Model 5.1 and above. Use `/enable_unbounded_descriptor_tables`.
EntryPointName	Optional **string** parameter. Specifies the name of the entry point for the shader. Use `/E[name]`.
GenerateExportFile	Optional **string** parameter.
GenerateExportShaderProfile	Optional **string** parameter.
HeaderFileOutput	Optional **string** parameter. Specifies a name for header file containing object code. Use `/Fh [name]`.
ObjectFileOutput	Optional **string** parameter. Specifies a name for object file. Use `/Fo [name]`.
PreprocessorDefinitions	Optional **string[]** parameter. Defines preprocessing symbols for your source file.
SetRootSignature	Optional **string** parameter. Attach root signature to shader bytecode. Available for Shader Model 5.0 and above. Use `/setrootsignature`.

PARAMETER	DESCRIPTION
ShaderModel	Optional **string** parameter. Specifies the shader model. Some shader types can only be used with recent shader models. Use `/T [type]_[model]`.
ShaderType	Optional **string** parameter. Specifies the type of shader. Use `/T [type]_[model]`. **Effect**, use `fx`. **Vertex**, use `vs`. **Pixel**, use `ps`. **Geometry**, use `gs`. **Hull**, use `hs`. **Domain**, use `ds`. **Compute**, use `cs`. **Library**, use `lib`. **RootSignature**, generate Root Signature Object.
Source	Required **ITaskItem** parameter.
SuppressStartupBanner	Optional **bool** parameter. Suppresses the display of the startup banner and information message. Use `/nologo`.
TrackerLogDirectory	Optional **string** parameter.
TreatWarningAsError	Optional **bool** parameter. Treats all compiler warnings as errors. For a new project, it may be best to use `/WX` in all compilations; resolving all warnings will ensure the fewest possible hard-to-find code defects.
VariableName	Optional **string** parameter. Specifies a name for the variable name in the header file. Use `/Vn [name]`.

See also

Task reference

GetOutOfDateItems task

10/21/2019 • 2 minutes to read • Edit Online

Helper task that reads old tlogs, writes new tlogs, and returns set of items that are not up-to-date.

Parameters

The following table describes the parameters of the **GetOutOfDateItems** task.

PARAMETER	DESCRIPTION
CheckForInterdependencies	Optional **bool** parameter.
CommandMetadataName	Optional **string** parameter.
DependenciesMetadataName	Optional **string** parameter.
HasInterdependencies	Optional **bool** output parameter.
OutOfDateSources	Optional **ITaskItem[]** output parameter.
OutputsMetadataName	Required **string** parameter.
Sources	Optional **ITaskItem[]** parameter.
TLogDirectory	Required **string** parameter.
TLogNamePrefix	Required **string** parameter.

See also

Task reference

GetOutputFileName task

10/21/2019 • 2 minutes to read • Edit Online

Helper task to get output file name for cl and other tools, which allow specifying only output directory or full file name or nothing.

Parameters

The following table describes the parameters of the **GetOutputFileName** task.

PARAMETER	DESCRIPTION
OutputExtension	Required **string** parameter.
OutputFile	Optional **string** output parameter.
OutputPath	Optional **string** parameter.
SourceFile	Required **string** parameter.

See also

Task reference

LIB task

10/21/2019 • 4 minutes to read • Edit Online

Wraps the Microsoft 32-Bit Library Manager tool, *lib.exe*. The Library Manager creates and manages a library of Common Object File Format (COFF) object files. The Library Manager can also create export files and import libraries to reference exported definitions. For more information, see LIB reference and Running LIB.

Parameters

The following table describes the parameters of the **LIB** task. Most task parameters correspond to a command-line option.

PARAMETER	DESCRIPTION
AdditionalDependencies	Optional **String[]** parameter. Specifies additional items to add to the command line.
AdditionalLibraryDirectories	Optional **String[]** parameter. Overrides the environment library path. Specify a directory name. For more information, see /LIBPATH (Additional Libpath).
AdditionalOptions	Optional **String** parameter. A list of *lib.exe* options as specified on the command line. For example, /<option1> /<option2> /<option#>. Use this parameter to specify *lib.exe* options that are not represented by any other **LIB** task parameter. For more information, see Running LIB.
DisplayLibrary	Optional **String** parameter. Displays information about the output library. Specify a file name to redirect the information to a file. Specify "CON" or nothing to redirect the information to the console. This parameter corresponds to the **/LIST** option of *lib.exe*.

PARAMETER	DESCRIPTION
ErrorReporting	Optional **String** parameter. Specifies how to send internal error information to Microsoft if *lib.exe* fails at run time. Specify one of the following values, each of which corresponds to a command-line option. - **NoErrorReport** - **/ERRORREPORT:NONE** - **PromptImmediately** - **/ERRORREPORT:PROMPT** - **QueueForNextLogin** - **/ERRORREPORT:QUEUE** - **SendErrorReport** - **/ERRORREPORT:SEND** For more information, see the **/ERRORREPORT** command-line option at Running LIB.
ExportNamedFunctions	Optional **String[]** parameter. Specifies one or more functions to export. This parameter corresponds to the **/EXPORT:** option of *lib.exe*.
ForceSymbolReferences	Optional **String** parameter. Forces *lib.exe* to include a reference to the specified symbol. This parameter corresponds to the **/INCLUDE:** option of *lib.exe*.
IgnoreAllDefaultLibraries	Optional `Boolean` parameter. If `true`, removes all default libraries from the list of libraries that *lib.exe* searches when it resolves external references. This parameter corresponds to the parameter-less form of the **/NODEFAULTLIB** option of *lib.exe*.
IgnoreSpecificDefaultLibraries	Optional **String[]** parameter. Removes the specified libraries from the list of libraries that *lib.exe* searches when it resolves external references. This parameter corresponds to the **/NODEFAULTLIB** option of *lib.exe* that takes a `library` argument.
LinkLibraryDependencies	Optional `Boolean` parameter. If `true`, specifies that library outputs from project dependencies are automatically linked in.
LinkTimeCodeGeneration	Optional `Boolean` parameter. If `true`, specifies link-time code generation. This parameter corresponds to the **/LCTG** option of *lib.exe*.

PARAMETER	DESCRIPTION
MinimumRequiredVersion	Optional **String** parameter. Specifies the minimum required version of the subsystem. Specify a comma-delimited list of decimal numbers in the range 0 through 65535.
ModuleDefinitionFile	Optional **String** parameter. Specifies the name of the module-definition file (*.def*). This parameter corresponds to the **/DEF** option of *lib.exe* that takes a `filename` argument.
Name	Optional **String** parameter. When an import library is built, specifies the name of the DLL for which the import library is being built. This parameter corresponds to the **/NAME** option of *lib.exe* that takes a `filename` argument.
OutputFile	Optional **String** parameter. Overrides the default name and location of the program that *lib.exe* creates. This parameter corresponds to the **/OUT** option of *lib.exe* that takes a `filename` argument.
RemoveObjects	Optional **String[]** parameter. Omits the specified object from the output library. *Lib.exe* creates an output library by combining all objects (whether in object files or libraries), and then deleting any objects that are specified by this option. This parameter corresponds to the **/REMOVE** option of *lib.exe* that takes a `membername` argument.
Sources	Required `ITaskItem[]` parameter. Specifies a list of source files separated by spaces.

PARAMETER	DESCRIPTION
SubSystem	Optional **String** parameter. Specifies the environment for the executable. The choice of subsystem affects the entry point symbol or entry point function. Specify one of the following values, each of which corresponds to a command-line option. - **Console** - **/SUBSYSTEM:CONSOLE** - **Windows** - **/SUBSYSTEM:WINDOWS** - **Native** - **/SUBSYSTEM:NATIVE** - **EFI Application** - **/SUBSYSTEM:EFI_APPLICATION** - **EFI Boot Service Driver** - **/SUBSYSTEM:EFI_BOOT_SERVICE_DRIVER** - **EFI ROM** - **/SUBSYSTEM:EFI_ROM** - **EFI Runtime** - **/SUBSYSTEM:EFI_RUNTIME_DRIVER** - **WindowsCE** - **/SUBSYSTEM:WINDOWSCE** - **POSIX** - **/SUBSYSTEM:POSIX** For more information, see /SUBSYSTEM (Specify subsystem).
SuppressStartupBanner	Optional **Boolean** parameter. If `true`, prevents the display of the copyright and version number message when the task starts. For more information, see the **/NOLOGO** option at Running LIB.
TargetMachine	Optional **String** parameter. Specifies the target platform for the program or DLL. Specify one of the following values, each of which corresponds to a command-line option. - **MachineARM** - **/MACHINE:ARM** - **MachineEBC** - **/MACHINE:EBC** - **MachineIA64** - **/MACHINE:IA64** - **MachineMIPS** - **/MACHINE:MIPS** - **MachineMIPS16** - **/MACHINE:MIPS16** - **MachineMIPSFPU** -**/MACHINE:MIPSFPU** - **MachineMIPSFPU16** - **/MACHINE:MIPSFPU16** - **MachineSH4** - **/MACHINE:SH4** - **MachineTHUMB** - **/MACHINE:THUMB** - **MachineX64** - **/MACHINE:X64** - **MachineX86** - **/MACHINE:X86** For more information, see /MACHINE (Specify target platform).
TrackerLogDirectory	Optional **String** parameter. Specifies the directory of the tracker log.

PARAMETER	DESCRIPTION
TreatLibWarningAsErrors	Optional **Boolean** parameter. If `true`, causes the **LIB** task to not generate an output file if *lib.exe* generates a warning. If `false`, an output file is generated. For more information, see the **/WX** option at Running LIB.
UseUnicodeResponseFiles	Optional **Boolean** parameter. If `true`, instructs the project system to generate UNICODE response files when the librarian is spawned. Specify `true` when files in the project have UNICODE paths.
Verbose	Optional **Boolean** parameter. If `true`, displays details about the progress of the session; this includes names of the *.obj* files being added. The information is sent to standard output and can be redirected to a file. For more information, see the **/VERBOSE** option in Running LIB.

See also

- Task reference

Link task

10/24/2019 • 15 minutes to read • Edit Online

Wraps the Microsoft C++ linker tool, *link.exe*. The linker tool links Common Object File Format (COFF) object files and libraries to create an executable (*.exe*) file or a dynamic-link library (DLL). For more information, see Linker options.

Parameters

The following describes the parameters of the **Link** task. Most task parameters, and a few sets of parameters, correspond to a command-line option.

- **AdditionalDependencies**

  Optional **String[]** parameter.

  Specifies a list of input files to add to the command.

  For more information, see LINK input files.

- **AdditionalLibraryDirectories**

  Optional **String[]** parameter.

  Overrides the environment library path. Specify a directory name.

  For more information, see /LIBPATH (Additional Libpath).

- **AdditionalManifestDependencies**

  Optional **String[]** parameter.

  Specifies attributes that will be placed in the `dependency` section of the manifest file.

  For more information, see /MANIFESTDEPENDENCY (Specify manifest dependencies). Also see Publisher configuration files.

- **AdditionalOptions**

  Optional **String** parameter.

  A list of linker options as specified on the command line. For example, /<option1> /<option2> /<option#>. Use this parameter to specify linker options that are not represented by any other **Link** task parameter.

  For more information, see Linker options.

- **AddModuleNamesToAssembly**

  Optional **String[]** parameter.

  Adds a module reference to an assembly.

  For more information, see /ASSEMBLYMODULE (Add a MSIL module to the assembly).

- **AllowIsolation**

  Optional **Boolean** parameter.

  If `true`, causes the operating system to do manifest lookups and loads. If `false`, indicates that DLLs are

loaded as if there was no manifest.

For more information, see /ALLOWISOLATION (Manifest lookup).

- **AssemblyDebug**

  Optional **Boolean** parameter.

  If `true`, emits the **DebuggableAttribute** attribute together with debug information tracking and disables JIT optimizations. If `false`, emits the **DebuggableAttribute** attribute but disables debug information tracking and enables JIT optimizations.

  For more information, see /ASSEMBLYDEBUG (Add DebuggableAttribute).

- **AssemblyLinkResource**

  Optional **String[]** parameter.

  Creates a link to a .NET Framework resource in the output file; the resource file is not placed in the output file. Specify the name of the resource.

  For more information, see /ASSEMBLYLINKRESOURCE (Link to .NET Framework resource).

- **AttributeFileTracking**

  Implicit **Boolean** parameter.

  Enables deeper file tracking to capture link incremental's behavior. Always returns `true`.

- **BaseAddress**

  Optional **String** parameter.

  Sets a base address for the program or DLL being built. Specify `{address[,size] | @filename,key}`.

  For more information, see /BASE (Base address).

- **BuildingInIDE**

  Optional **Boolean** parameter.

  If true, indicates that MSBuild is invoked from the IDE. Otherwise, indicates that MSBuild is invoked from the command line.

  This parameter has no equivalent linker option.

- **CLRImageType**

  Optional **String** parameter.

  Sets the type of a common language runtime (CLR) image.

  Specify one of the following values, each of which corresponds to a linker option.

  - **Default** - *<none>*
  - **ForceIJWImage** - **/CLRIMAGETYPE:IJW**
  - **ForcePureILImage** - **/CLRIMAGETYPE:PURE**
  - **ForceSafeILImage** - **/CLRIMAGETYPE:SAFE**

  For more information, see /CLRIMAGETYPE (Specify type of CLR image).

- **CLRSupportLastError**

Optional **String** parameter.

Preserves the last error code of functions called through the P/Invoke mechanism.

Specify one of the following values, each of which corresponds to a linker option.

- **Enabled** - **/CLRSupportLastError**
- **Disabled** - **/CLRSupportLastError:NO**
- **SystemDlls** - **/CLRSupportLastError:SYSTEMDLL**

For more information, see /CLRSUPPORTLASTERROR (Preserve last error code for PInvoke calls).

- **CLRThreadAttribute**

  Optional **String** parameter.

  Explicitly specifies the threading attribute for the entry point of your CLR program.

  Specify one of the following values, each of which corresponds to a linker option.

  - **DefaultThreadingAttribute** - **/CLRTHREADATTRIBUTE:NONE**
  - **MTAThreadingAttribute** - **/CLRTHREADATTRIBUTE:MTA**
  - **STAThreadingAttribute** - **/CLRTHREADATTRIBUTE:STA**

  For more information, see /CLRTHREADATTRIBUTE (Set CLR thread attribute).

- **CLRUnmanagedCodeCheck**

  Optional **Boolean** parameter.

  Specifies whether the linker will apply **SuppressUnmanagedCodeSecurityAttribute** to linker-generated P/Invoke calls from managed code into native DLLs.

  For more information, see /CLRUNMANAGEDCODECHECK (Add SuppressUnmanagedCodeSecurityAttribute).

- **CreateHotPatchableImage**

  Optional **String** parameter.

  Prepares an image for hot patching.

  Specify one of the following values, which corresponds to a linker option.

  - **Enabled** - **/FUNCTIONPADMIN**
  - **X86Image** - **/FUNCTIONPADMIN:5**
  - **X64Image** - **/FUNCTIONPADMIN:6**
  - **ItaniumImage** - **/FUNCTIONPADMIN:16**

  For more information, see /FUNCTIONPADMIN (Create hotpatchable image).

- **DataExecutionPrevention**

  Optional **Boolean** parameter.

  If `true`, indicates that an executable was tested to be compatible with the Windows Data Execution Prevention feature.

For more information, see /NXCOMPAT (Compatible with Data Execution Prevention).

- **DelayLoadDLLs**

  Optional **String[]** parameter.

  This parameter causes *delayed loading* of DLLs. Specify the name of a DLL to delay load.

  For more information, see /DELAYLOAD (Delay load import).

- **DelaySign**

  Optional **Boolean** parameter.

  If `true`, partially signs an assembly. By default, the value is `false`.

  For more information, see /DELAYSIGN (Partially sign an assembly).

- **Driver**

  Optional **String** parameter.

  Specify this parameter to build a Windows NT kernel mode driver.

  Specify one of the following values, each of which corresponds to a linker option.

  - **NotSet** - *<none>*
  - **Driver** - **/Driver**
  - **UpOnly** - **/DRIVER:UPONLY**
  - **WDM** - **/DRIVER:WDM**

  For more information, see /DRIVER (Windows NT kernel mode driver).

- **EmbedManagedResourceFile**

  Optional **String[]** parameter.

  Embeds a resource file in an assembly. Specify the required resource file name. Optionally specify the logical name, which is used to load the resource, and the **PRIVATE** option, which indicates in the assembly manifest that the resource file is private.

  For more information, see /ASSEMBLYRESOURCE (Embed a managed resource).

- **EnableCOMDATFolding**

  Optional **Boolean** parameter.

  If `true`, enables identical COMDAT folding.

  For more information, see the `ICF[= iterations]` argument of /OPT (Optimizations).

- **EnableUAC**

  Optional **Boolean** parameter.

  If `true`, specifies that User Account Control (UAC) information is embedded in the program manifest.

  For more information, see /MANIFESTUAC (Embeds UAC information in manifest).

- **EntryPointSymbol**

  Optional **String** parameter.

Specifies an entry point function as the starting address for an *.exe* file or DLL. Specify a function name as the parameter value.

For more information, see /ENTRY (Entry-point symbol).

- **FixedBaseAddress**

  Optional **Boolean** parameter.

  If `true`, creates a program or DLL that can be loaded only at its preferred base address.

  For more information, see /FIXED (Fixed base address).

- **ForceFileOutput**

  Optional **String** parameter.

  Tells the linker to create a valid *.exe* file or DLL even if a symbol is referenced but not defined, or is multiply defined.

  Specify one of the following values, each of which corresponds to a command-line option.

  - **Enabled** - **/FORCE**
  - **MultiplyDefinedSymbolOnly** - **/FORCE:MULTIPLE**
  - **UndefinedSymbolOnly** - **/FORCE:UNRESOLVED**

  For more information, see /FORCE (Force file output).

- **ForceSymbolReferences**

  Optional **String[]** parameter.

  This parameter tells the linker to add a specified symbol to the symbol table.

  For more information, see /INCLUDE (Force symbol references).

- **FunctionOrder**

  Optional **String** parameter.

  This parameter optimizes your program by placing the specified packaged functions (COMDATs) into the image in a predetermined order.

  For more information, see /ORDER (Put functions in order).

- **GenerateDebugInformation**

  Optional **Boolean** parameter.

  If `true`, creates debugging information for the *.exe* file or DLL.

  For more information, see /DEBUG (Generate debug info).

- **GenerateManifest**

  Optional **Boolean** parameter.

  If `true`, creates a side-by-side manifest file.

  For more information, see /MANIFEST (Create side-by-side assembly manifest).

- **GenerateMapFile**

Optional **Boolean** parameter.

If `true`, creates a *map file*. The file name extension of the map file is *.map*.

For more information, see /MAP (Generate mapfile).

- **HeapCommitSize**

  Optional **String** parameter.

  Specifies the amount of physical memory on the heap to allocate at a time.

  For more information, see the `commit` argument in /HEAP (Set heap size). Also, see the **HeapReserveSize** parameter.

- **HeapReserveSize**

  Optional **String** parameter.

  Specifies the total heap allocation in virtual memory.

  For more information, see the `reserve` argument in /HEAP (Set heap size). Also, see the **HeapCommitSize** parameter in this table.

- **IgnoreAllDefaultLibraries**

  Optional **Boolean** parameter.

  If `true`, tells the linker to remove one or more default libraries from the list of libraries it searches when it resolves external references.

  For more information, see /NODEFAULTLIB (Ignore libraries).

- **IgnoreEmbeddedIDL**

  Optional **Boolean** parameter.

  If `true`, specifies that any IDL attributes in source code should not be processed into an *.idl* file.

  For more information, see /IGNOREIDL (Don't process attributes into MIDL).

- **IgnoreImportLibrary**

  Optional **Boolean** parameter.

  If `true`, specifies that the import library generated by this configuration should not be imported into dependent projects.

  This parameter does not correspond to a linker option.

- **IgnoreSpecificDefaultLibraries**

  Optional **String[]** parameter.

  Specifies one or more names of default libraries to ignore. Separate multiple libraries by using semi-colons.

  For more information, see /NODEFAULTLIB (Ignore libraries).

- **ImageHasSafeExceptionHandlers**

  Optional **Boolean** parameter.

  If `true`, the linker produces an image only if it can also produce a table of the image's safe exception handlers.

For more information, see /SAFESEH (Image has safe exception handlers).

- **ImportLibrary**

  A user-specified import library name that replaces the default library name.

  For more information, see /IMPLIB (Name import library).

- **KeyContainer**

  Optional **String** parameter.

  Container that contains the key for a signed assembly.

  For more information, see /KEYCONTAINER (Specify a key container to sign an assembly). Also, see the **KeyFile** parameter in this table.

- **KeyFile**

  Optional **String** parameter.

  Specifies a file that contains the key for a signed assembly.

  For more information, see /KEYFILE (Specify key or key pair to sign an assembly). Also, see the **KeyContainer** parameter.

- **LargeAddressAware**

  Optional **Boolean** parameter.

  If `true`, the application can handle addresses larger than 2 gigabytes.

  For more information, see /LARGEADDRESSAWARE (Handle large addresses).

- **LinkDLL**

  Optional **Boolean** parameter.

  If `true`, builds a DLL as the main output file.

  For more information, see /DLL (Build a DLL).

- **LinkErrorReporting**

  Optional **String** parameter.

  Lets you provide internal compiler error (ICE) information directly to Microsoft.

  Specify one of the following values, each of which corresponds to a command-line option.

  - **NoErrorReport** - **/ERRORREPORT:NONE**
  - **PromptImmediately** - **/ERRORREPORT:PROMPT**
  - **QueueForNextLogin** - **/ERRORREPORT:QUEUE**
  - **SendErrorReport** - **/ERRORREPORT:SEND**

  For more information, see /ERRORREPORT (Report internal linker errors).

- **LinkIncremental**

  Optional **Boolean** parameter.

  If `true`, enables incremental linking.

For more information, see /INCREMENTAL (Link incrementally).

- **LinkLibraryDependencies**

  Optional **Boolean** parameter.

  If `true`, specifies that library outputs from project dependencies are automatically linked in.

  This parameter does not correspond to a linker option.

- **LinkStatus**

  Optional **Boolean** parameter.

  If `true`, specifies that the linker is to display a progress indicator that shows what percentage of the link is complete.

  For more information, see the `STATUS` argument of /LTCG (Link-time code generation).

- **LinkTimeCodeGeneration**

  Optional **String** parameter.

  Specifies options for profile-guided optimization.

  Specify one of the following values, each of which corresponds to a command-line option.

  - **Default** - *<none>*
  - **UseLinkTimeCodeGeneration** - **/LTCG**
  - **PGInstrument** - **/LTCG:PGInstrument**
  - **PGOptimization** - **/LTCG:PGOptimize**
  - **PGUpdate**

    - **/LTCG:PGUpdate**

  For more information, see /LTCG (Link-time code generation).

- **ManifestFile**

  Optional **String** parameter.

  Changes the default manifest file name to the specified file name.

  For more information, see /MANIFESTFILE (Name manifest file).

- **MapExports**

  Optional **Boolean** parameter.

  If `true`, tells the linker to include exported functions in a map file.

  For more information, see the `EXPORTS` argument of /MAPINFO (Include information in mapfile).

- **MapFileName**

  Optional **String** parameter.

  Changes the default map file name to the specified file name.

- **MergedIDLBaseFileName**

  Optional **String** parameter.

Specifies the file name and file name extension of the *.idl* file.

For more information, see /IDLOUT (Name MIDL output files).

- **MergeSections**

  Optional **String** parameter.

  Combines sections in an image. Specify `from-section=to-section`.

  For more information, see /MERGE (Combine sections).

- **MidlCommandFile**

  Optional **String** parameter.

  Specify the name of a file that contains MIDL command-line options.

  For more information, see /MIDL (Specify MIDL command line options).

- **MinimumRequiredVersion**

  Optional **String** parameter.

  Specifies the minimum required version of the subsystem. The arguments are decimal numbers in the range 0 through 65535.

- **ModuleDefinitionFile**

  Optional **String** parameter.

  Specifies the name of a module definition file.

  For more information, see /DEF (Specify module-definition file).

- **MSDOSStubFileName**

  Optional **String** parameter.

  Attaches the specified MS-DOS stub program to a Win32 program.

  For more information, see /STUB (MS-DOS stub file name).

- **NoEntryPoint**

  Optional **Boolean** parameter.

  If `true`, specifies a resource-only DLL.

  For more information, see /NOENTRY (No entry point).

- **ObjectFiles**

  Implicit **String[]** parameter.

  Specifies the object files that are linked.

- **OptimizeReferences**

  Optional **Boolean** parameter.

  If `true`, eliminates functions and/or data that are never referenced.

  For more information, see the `REF` argument in /OPT (Optimizations).

- **OutputFile**

Optional **String** parameter.

Overrides the default name and location of the program that the linker creates.

For more information, see /OUT (Output file name).

- **PerUserRedirection**

  Optional **Boolean** parameter.

  If `true` and Register Output is enabled, forces registry writes to **HKEY_CLASSES_ROOT** to be redirected to **HKEY_CURRENT_USER**.

- **PreprocessOutput**

  Optional `ITaskItem[]` parameter.

  Defines an array of preprocessor output items that can be consumed and emitted by tasks.

- **PreventDllBinding**

  Optional **Boolean** parameter.

  If `true`, indicates to *Bind.exe* that the linked image should not be bound.

  For more information, see /ALLOWBIND (Prevent DLL binding).

- **Profile**

  Optional **Boolean** parameter.

  If `true`, produces an output file that can be used with the **Performance Tools** profiler.

  For more information, see /PROFILE (Performance Tools profiler).

- **ProfileGuidedDatabase**

  Optional **String** parameter.

  Specifies the name of the *.pgd* file that will be used to hold information about the running program

  For more information, see /PGD (Specify database for profile-guided optimizations).

- **ProgramDatabaseFile**

  Optional **String** parameter.

  Specifies a name for the program database (PDB) that the linker creates.

  For more information, see /PDB (Use program database).

- **RandomizedBaseAddress**

  Optional **Boolean** parameter.

  If `true`, generates an executable image that can be randomly rebased at load time by using the *address space layout randomization* (ASLR) feature of Windows.

  For more information, see /DYNAMICBASE (Use address space layout randomization).

- **RegisterOutput**

  Optional **Boolean** parameter.

  If `true`, registers the primary output of this build.

- **SectionAlignment**

  Optional **Integer** parameter.

  Specifies the alignment of each section within the linear address space of the program. The parameter value is a unit number of bytes and is a power of two.

  For more information, see /ALIGN (Section alignment).

- **SetChecksum**

  Optional **Boolean** parameter.

  If `true`, sets the checksum in the header of an *.exe* file.

  For more information, see /RELEASE (Set the checksum).

- **ShowProgress**

  Optional **String** parameter.

  Specifies the verbosity of progress reports for the linking operation.

  Specify one of the following values, each of which corresponds to a command-line option.

  - **NotSet** - *<none>*
  - **LinkVerbose** - **/VERBOSE**
  - **LinkVerboseLib** - **/VERBOSE:Lib**
  - **LinkVerboseICF** - **/VERBOSE:ICF**
  - **LinkVerboseREF** - **/VERBOSE:REF**
  - **LinkVerboseSAFESEH** - **/VERBOSE:SAFESEH**
  - **LinkVerboseCLR** - **/VERBOSE:CLR**

  For more information, see /VERBOSE (Print progress messages).

- **Sources**

  Required `ITaskItem[]` parameter.

  Defines an array of MSBuild source file items that can be consumed and emitted by tasks.

- **SpecifySectionAttributes**

  Optional **String** parameter.

  Specifies the attributes of a section. This overrides the attributes that were set when the *.obj* file for the section was compiled.

  For more information, see /SECTION (Specify section attributes).

- **StackCommitSize**

  Optional **String** parameter.

  Specifies the amount of physical memory in each allocation when additional memory is allocated.

  For more information, see the `commit` argument of /STACK (Stack allocations).

- **StackReserveSize**

Optional **String** parameter.

Specifies the total stack allocation size in virtual memory.

For more information, see the `reserve` argument of /STACK (Stack allocations).

- **StripPrivateSymbols**

  Optional **String** parameter.

  Creates a second program database (PDB) file that omits symbols that you do not want to distribute to your customers. Specify the name of the second PDB file.

  For more information, see /PDBSTRIPPED (Strip private symbols).

- **SubSystem**

  Optional **String** parameter.

  Specifies the environment for the executable.

  Specify one of the following values, each of which corresponds to a command-line option.

  - **NotSet** - *<none>*
  - **Console** - **/SUBSYSTEM:CONSOLE**
  - **Windows** - **/SUBSYSTEM:WINDOWS**
  - **Native** - **/SUBSYSTEM:NATIVE**
  - **EFI Application** - **/SUBSYSTEM:EFI_APPLICATION**
  - **EFI Boot Service Driver** - **/SUBSYSTEM:EFI_BOOT_SERVICE_DRIVER**
  - **EFI ROM** - **/SUBSYSTEM:EFI_ROM**
  - **EFI Runtime** - **/SUBSYSTEM:EFI_RUNTIME_DRIVER**
  - **WindowsCE** - **/SUBSYSTEM:WINDOWSCE**
  - **POSIX** - **/SUBSYSTEM:POSIX**

  For more information, see /SUBSYSTEM (Specify subsystem).

- **SupportNobindOfDelayLoadedDLL**

  Optional **Boolean** parameter.

  If `true`, tells the linker not to include a bindable Import Address Table (IAT) in the final image.

  For more information, see the `NOBIND` argument of /DELAY (Delay load import settings).

- **SupportUnloadOfDelayLoadedDLL**

  Optional **Boolean** parameter.

  If `true`, tells the delay-load helper function to support explicit unloading of the DLL.

  For more information, see the `UNLOAD` argument of /DELAY (Delay load import settings).

- **SuppressStartupBanner**

  Optional **Boolean** parameter.

  If `true`, prevents the display of the copyright and version number message when the task starts.

For more information, see /NOLOGO (Suppress startup banner) (linker).

- **SwapRunFromCD**

  Optional **Boolean** parameter.

  If `true`, tells the operating system to first copy the linker output to a swap file, and then run the image from there.

  For more information, see the `CD` argument of /SWAPRUN (Load linker output to swap file). Also, see the **SwapRunFromNET** parameter.

- **SwapRunFromNET**

  Optional **Boolean** parameter.

  If `true`, tells the operating system to first copy the linker output to a swap file, and then run the image from there.

  For more information, see the `NET` argument of /SWAPRUN (Load linker output to swap file). Also, see the **SwapRunFromCD** parameter in this table.

- **TargetMachine**

  Optional **String** parameter.

  Specifies the target platform for the program or DLL.

  Specify one of the following values, each of which corresponds to a command-line option.

  - **NotSet** - *<none>*
  - **MachineARM** - **/MACHINE:ARM**
  - **MachineEBC** - **/MACHINE:EBC**
  - **MachineIA64** - **/MACHINE:IA64**
  - **MachineMIPS** - **/MACHINE:MIPS**
  - **MachineMIPS16** - **/MACHINE:MIPS16**
  - **MachineMIPSFPU** - **/MACHINE:MIPSFPU**
  - **MachineMIPSFPU16** - **/MACHINE:MIPSFPU16**
  - **MachineSH4** - **/MACHINE:SH4**
  - **MachineTHUMB** - **/MACHINE:THUMB**
  - **MachineX64** - **/MACHINE:X64**
  - **MachineX86** - **/MACHINE:X86**

  For more information, see /MACHINE (Specify target platform).

- **TerminalServerAware**

  Optional **Boolean** parameter.

  If `true`, sets a flag in the IMAGE_OPTIONAL_HEADER DllCharacteristics field in the program image's optional header. When this flag is set, Terminal Server will not make certain changes to the application.

  For more information, see /TSAWARE (Create Terminal Server aware application).

- **TrackerLogDirectory**

  Optional **String** parameter.

  Specifies the directory of the tracker log.

- **TreatLinkerWarningAsErrors**

  Optional **Boolean** parameter.

  If `true`, causes no output file to be generated if the linker generates a warning.

  For more information, see /WX (Treat linker warnings as errors).

- **TurnOffAssemblyGeneration**

  Optional **Boolean** parameter.

  If `true`, creates an image for the current output file without a .NET Framework assembly.

  For more information, see /NOASSEMBLY (Create a MSIL module).

- **TypeLibraryFile**

  Optional **String** parameter.

  Specifies the file name and file name extension of the *.tlb* file. Specify a file name, or a path and file name.

  For more information, see /TLBOUT (Name .tlb file).

- **TypeLibraryResourceID**

  Optional **Integer** parameter.

  Designates a user-specified value for a linker-created type library. Specify a value from 1 through 65535.

  For more information, see /TLBID (Specify resource ID for TypeLib).

- **UACExecutionLevel**

  Optional **String** parameter.

  Specifies the requested execution level for the application when it is run under with User Account Control.

  Specify one of the following values, each of which corresponds to a command-line option.

  - **AsInvoker** - `level='asInvoker'`
  - **HighestAvailable** - `level='highestAvailable'`
  - **RequireAdministrator** - `level='requireAdministrator'`

  For more information, see the `level` argument of /MANIFESTUAC (Embeds UAC information in manifest).

- **UACUIAccess**

  Optional **Boolean** parameter.

  If `true`, the application bypasses user interface protection levels and drives input to higher-permission windows on the desktop; otherwise, `false`.

  For more information, see the `uiAccess` argument of /MANIFESTUAC (Embeds UAC information in manifest).

- **UseLibraryDependencyInputs**

Optional **Boolean** parameter.

If `true`, the inputs to the librarian tool are used rather than the library file itself when library outputs of project dependencies are linked in.

- **Version**

  Optional **String** parameter.

  Put a version number in the header of the *.exe* or *.dll* file. Specify "`major[.minor]`". The `major` and `minor` arguments are decimal numbers from 0 through 65535.

  For more information, see /VERSION (Version information).

See also

- Task reference

MIDL task

10/24/2019 • 7 minutes to read • Edit Online

Wraps the Microsoft Interface Definition Language (MIDL) compiler tool, *midl.exe*. For more information, see MIDL command-line reference.

Parameters

The following describes the parameters of the **MIDL** task. Most task parameters, and a few sets of parameters, correspond to a command-line option.

- **AdditionalIncludeDirectories**

  Optional **String[]** parameter.

  Adds a directory to the list of directories that are searched for imported IDL files, included header files, and application configuration files (ACF).

  For more information, see the **/I** option in MIDL command-line reference.

- **AdditionalOptions**

  Optional **String** parameter.

  A list of command-line options. For example, /<option1> /<option2> /<option#>. Use this parameter to specify command-line options that are not represented by any other MIDL task parameter.

  For more information, see MIDL command-line reference.

- **ApplicationConfigurationMode**

  Optional **Boolean** parameter.

  If `true`, lets you use some ACF keywords in the IDL file.

  For more information, see the **/app_config** option in MIDL command-line reference.

- **ClientStubFile**

  Optional **String** parameter.

  Specifies the name of the client stub file for an RPC interface.

  For more information, see the **/cstub** option in MIDL command-line reference. Also see the **ServerStubFile** parameter in this table.

- **CPreprocessOptions**

  Optional **String** parameter.

  Specifies options to pass to the C/C++ preprocessor. Specify a space-delimited list of preprocessor options.

  For more information, see the **/cpp_opt** option in MIDL command-line reference.

- **DefaultCharType**

  Optional **String** parameter.

  Specifies the default character type that the C compiler will use to compile the generated code.

Specify one of the following values, each of which corresponds to a command-line option.

VALUE	COMMAND-LINE OPTION
Signed	**/char signed**
Unsigned	**/char unsigned**
Ascii	**/char ascii7**

For more information, see the **/char** option in MIDL command-line reference.

- **DllDataFileName**

  Optional **String** parameter.

  Specifies the file name for the generated *dlldata* file for a proxy DLL.

  For more information, see the **/dlldata** option in MIDL command-line reference.

- **EnableErrorChecks**

  Optional **String** parameter.

  Specifies the type of error checking that the generated stubs will perform at run time.

  Specify one of the following values, each of which corresponds to a command-line option.

VALUE	COMMAND-LINE OPTION
None	**/error none**
EnableCustom	**/error**
All	**/error all**

  For more information, see the **/error** option in MIDL command-line reference.

- **ErrorCheckAllocations**

  Optional **Boolean** parameter.

  If `true`, check for out-of-memory errors.

  For more information, see the **/error allocation** option in MIDL command-line reference.

- **ErrorCheckBounds**

  Optional **Boolean** parameter.

  If `true`, checks the size of conformant-varying and varying arrays against the transmission length specification.

  For more information, see the **/error bounds_check** option in MIDL command-line reference.

- **ErrorCheckEnumRange**

  Optional **Boolean** parameter.

  If `true`, checks that enum values are in an allowable range.

For more information, see the **/error enum** option in command-line help (**/?**) for *midl.exe*.

- **ErrorCheckRefPointers**

  Optional **Boolean** parameter.

  If `true`, check that no null reference pointers are passed to client stubs.

  For more information, see the **/error ref** option in MIDL command-line reference.

- **ErrorCheckStubData**

  Optional **Boolean** parameter.

  If `true`, generates a stub that catches unmarshaling exceptions on the server side and propagates them back to the client.

  For more information, see the **/error stub_data** option in MIDL command-line reference.

- **GenerateClientFiles**

  Optional **String** parameter.

  Specifies whether the compiler generates client-side C source files for an RPC interface.

  Specify one of the following values, each of which corresponds to a command-line option.

VALUE	COMMAND-LINE OPTION
None	**/client none**
Stub	**/client stub**

  For more information, see the **/client** option in MIDL command-line reference.

- **GenerateServerFiles**

  Optional **String** parameter.

  Specifies whether the compiler generates server-side C source files for an RPC interface.

  Specify one of the following values, each of which corresponds to a command-line option.

VALUE	COMMAND-LINE OPTION
None	**/server none**
Stub	**/server stub**

  For more information, see the **/server** option in MIDL command-line reference.

- **GenerateStublessProxies**

  Optional **Boolean** parameter.

  If `true`, generates fully interpreted stubs together with stubless proxies for object interfaces.

  For more information, see the **/Oicf** option in MIDL command-line reference.

- **GenerateTypeLibrary**

  Optional **Boolean** parameter.

If `true`, a type library (*.tlb*) file is not generated.

For more information, see the **/notlb** option in MIDL command-line reference.

- **HeaderFileName**

  Optional **String** parameter.

  Specifies the name of the generated header file.

  For more information, see the **/h** or **/header** option in MIDL command-line reference.

- **IgnoreStandardIncludePath**

  Optional **Boolean** parameter.

  If `true`, the MIDL task searches only the directories specified by using the **AdditionalIncludeDirectories** switch, and ignores the current directory and the directories specified by the INCLUDE environment variable.

  For more information, see the **/no_def_idir** option in MIDL command-line reference.

- **InterfaceIdentifierFileName**

  Optional **String** parameter.

  Specifies the name of the *interface identifier file* for a COM interface. This overrides the default name obtained by adding "_i.c" to the IDL file name.

  For more information, see the **/iid** option in MIDL command-line reference.

- **LocaleID**

  Optional **int** parameter.

  Specifies the *locale identifier* that enables the use of international characters in input files, file names, and directory paths. Specify a decimal locale identifier.

  For more information, see the **/lcid** option in MIDL command-line reference. Also see Locale identifiers.

- **MkTypLibCompatible**

  Optional **Boolean** parameter.

  If `true`, requires the format of the input file to be compatible with *mktyplib.exe* version 2.03.

  For more information, see the **/mktyplib203** option in MIDL command-line reference. Also, see ODL file syntax on the MSDN website.

- **OutputDirectory**

  Optional **String** parameter.

  Specifies the default directory where the MIDL task writes output files.

  For more information, see the **/out** option in MIDL command-line reference.

- **PreprocessorDefinitions**

  Optional **String[]** parameter.

  Specifies one or more *defines*; that is, a name and an optional value to be passed to the C preprocessor as if by a `#define` directive. The form of each define is, *name[=value]*.

For more information, see the **/D** option in MIDL command-line reference. Also, see the **UndefinePreprocessorDefinitions** parameter in this table.

- **ProxyFileName**

  Optional **String** parameter.

  Specifies the name of the interface proxy file for a COM interface.

  For more information, see the **/proxy** option in MIDL command-line reference.

- **RedirectOutputAndErrors**

  Optional **String** parameter.

  Redirects output, such as error messages and warnings, from standard output to the specified file.

  For more information, see the **/o** option in MIDL command-line reference.

- **ServerStubFile**

  Optional **String** parameter.

  Specifies the name of the server stub file for an RPC interface.

  For more information, see the **/sstub** option in MIDL command-line reference. Also, see the **ClientStubFile** parameter in this table.

- **Source**

  Required `ITaskItem[]` parameter.

  Specifies a list of source files separated by spaces.

- **StructMemberAlignment**

  Optional **String** parameter.

  Specifies the alignment (*packing level*) of structures in the target system.

  Specify one of the following values, each of which corresponds to a command-line option.

VALUE	COMMAND-LINE OPTION
NotSet	*<none>*
1	**/Zp1**
2	**/Zp2**
4	**/Zp4**
8	**/Zp8**

  For more information, see the **/Zp** option in MIDL command-line reference. The **/Zp** option is equivalent to the **/pack** option and the older **/align** option.

- **SuppressCompilerWarnings**

  Optional **Boolean** parameter.

  If `true`, suppresses warning messages from the MIDL task.

For more information, see the **/no_warn** option in MIDL command-line reference.

- **SuppressStartupBanner**

  Optional `Boolean` parameter.

  If `true`, prevents the display of the copyright and version number message when the task starts.

  For more information, see the **/nologo** option in MIDL command-line reference.

- **TargetEnvironment**

  Optional **String** parameter.

  Specifies the environment in which the application runs.

  Specify one of the following values, each of which corresponds to a command-line option.

VALUE	COMMAND-LINE OPTION
NotSet	*<none>*
Win32	**/env win32**
Itanium	**/env ia64**
X64	**/env x64**

  For more information, see the **/env** option in MIDL command-line reference.

- **TrackerLogDirectory**

  Optional `String` parameter.

  Specifies the intermediate directory where tracking logs for this task are stored.

- **TypeLibFormat**

  Optional **String** parameter.

  Specifies the format of the type library file.

  Specify one of the following values, each of which corresponds to a command-line option.

VALUE	COMMAND-LINE OPTION
NewFormat	**/newtlb**
OldFormat	**/oldtlb**

  For more information, see the **/newtlb** and **/oldtlb** options in MIDL command-line reference.

- **TypeLibraryName**

  Optional **String** parameter.

  Specifies the name of the type library file.

  For more information, see the **/tlb** option in MIDL command-line reference.

- **UndefinePreprocessorDefinitions**

Optional **String[]** parameter.

Removes any previous definition of a name by passing the name to the C preprocessor as if by a `#undefine` directive. Specify one or more previously defined names.

For more information, see the **/U** option in MIDL command-line reference. Also, see the **PreprocessorDefinitions** parameter in this table.

- **ValidateAllParameters**

  Optional `Boolean` parameter.

  If `true`, generates additional error-checking information that is used to perform integrity checks at run time. If `false`, the error-checking information is not generated.

  For more information, see the **/robust** and **/no_robust** options in MIDL command-line reference.

- **WarnAsError**

  Optional `Boolean` parameter.

  If `true`, treats all warnings as errors.

  If the **WarningLevel** MIDL task parameter is not specified, warnings at the default level, level 1, are treated as errors.

  For more information, see the **/WX** options in MIDL command-line reference. Also, see the **WarningLevel** parameter in this table.

- **WarningLevel**

  Optional **String** parameter.

  Specifies the severity (*warning level*) of warnings to emit. No warning is emitted for a value of 0. Otherwise, a warning is emitted if its warning level is numerically less than or equal to the specified value.

  Specify one of the following values, each of which corresponds to a command-line option.

VALUE	COMMAND-LINE OPTION
0	**/W0**
1	**/W1**
2	**/W2**
3	**/W3**
4	**/W4**

  For more information, see the **/W** option in MIDL command-line reference. Also, see the **WarnAsError** parameter in this table.

See also

- Task reference

MT task

10/24/2019 • 4 minutes to read • Edit Online

Wraps the Microsoft Manifest Tool, *mt.exe*. For more information, see Mt.exe.

Parameters

The following table describes the parameters of the **MT** task. Most task parameters, and a few sets of parameters, correspond to a command-line option.

> **NOTE**
>
> The *mt.exe* documentation uses a hyphen (-) as the prefix for command-line options, but this topic uses a slash (/). Either prefix is acceptable.

PARAMETER	DESCRIPTION
AdditionalManifestFiles	Optional **String[]** parameter. Specifies the name of one or more manifest files. For more information, see the **/manifest** option in Mt.exe.
AdditionalOptions	Optional **String** parameter. A list of command-line options. For example, /<option1> /<option2> /<option#>. Use this parameter to specify command-line options that are not represented by any other **MT** task parameter. For more information, see Mt.exe.
AssemblyIdentity	Optional **String** parameter. Specifies the attribute values of the **assemblyIdentity** element of the manifest. Specify a comma-delimited list, where the first component is the value of the `name` attribute, followed by one or more name/value pairs that have the form, *<attribute name>=<attribute_value>*. For more information, see the **/identity** option in Mt.exe.
ComponentFileName	Optional **String** parameter. Specifies the name of the dynamic-link library you intend to build from the *.rgs* or *.tlb* files. This parameter is required if you specify the **RegistrarScriptFile** or **TypeLibraryFile** MT task parameters. For more information, see the **/dll** option in Mt.exe.

PARAMETER	DESCRIPTION
DependencyInformationFile	Optional **String** parameter. Specifies the dependency information file used by Visual Studio to track build dependency information for the manifest tool.
EmbedManifest	Optional `Boolean` parameter. If `true`, embeds the manifest file in the assembly. If `false`, creates as a stand-alone manifest file.
EnableDPIAwareness	Optional `Boolean` parameter. If `true`, adds to the manifest information that marks the application as DPI-aware. Writing a DPI-aware application makes a user interface look consistently good across a wide variety of high-DPI display settings. For more information, see High DPI.
GenerateCatalogFiles	Optional `Boolean` parameter. If `true`, generates catalog definition (*.cdf*) files. For more information, see the **/makecdfs** option in Mt.exe.
GenerateCategoryTags	Optional `Boolean` parameter. If `true`, causes category tags to be generated. If this parameter is `true`, the **ManifestFromManagedAssemblyMT** task parameter must also be specified. For more information, see the **/category** option in Mt.exe.
InputResourceManifests	Optional **String** parameter. Input the manifest from a resource of type RT_MANIFEST that has the specified identifier. Specify a resource of the form, <file>[;[#]<resource_id>], where the optional <resource_id> parameter is a non-negative, 16-bit number. If no `resource_id` is specified, the CREATEPROCESS_MANIFEST_RESOURCE default value (1) is used. For more information, see the **/inputresource** option in Mt.exe.
ManifestFromManagedAssembly	Optional **String** parameter. Generates a manifest from the specified managed assembly. For more information, see the **/managedassemblyname** option in Mt.exe.

PARAMETER	DESCRIPTION
ManifestToIgnore	Optional **String** parameter. (Not used.)
OutputManifestFile	Optional **String** parameter. Specifies the name of the output manifest. If this parameter is omitted and only one manifest is being operated on, that manifest is modified in place. For more information, see the **/out** option in Mt.exe.
OutputResourceManifests	Optional **String** parameter. Output the manifest to a resource of type RT_MANIFEST that has the specified identifier. The resource is of the form, <file>[;[#]<resource_id>], where the optional <resource_id> parameter is a non-negative, 16-bit number. If no `resource_id` is specified, the CREATEPROCESS_MANIFEST_RESOURCE default value (1) is used. For more information, see the **/outputresource** option in Mt.exe.
RegistrarScriptFile	Optional **String** parameter. Specifies the name of the registrar script (*.rgs*) file to use for registration-free COM manifest support. For more information, see the **/rgs** option in Mt.exe.
ReplacementsFile	Optional **String** parameter. Specifies the file that contains values for the replaceable strings in the registrar script (*.rgs*) file. For more information, see the **/replacements** option in Mt.exe.
ResourceOutputFileName	Optional **String** parameter. Specifies the output resources file used to embed the manifest into the project output.
Sources	Optional `ITaskItem[]` parameter. Specifies a list of manifest source files separated by spaces. For more information, see the **/manifest** option in Mt.exe.

PARAMETER	DESCRIPTION
SuppressDependencyElement	Optional `Boolean` parameter. If `true`, generates a manifest without dependency elements. If this parameter is `true`, also specify the **ManifestFromManagedAssemblyMT** task parameter. For more information, see the **/nodependency** option in Mt.exe.
SuppressStartupBanner	Optional `Boolean` parameter. If `true`, prevents the display of the copyright and version number message when the task starts. For more information, see the **/nologo** option in Mt.exe.
TrackerLogDirectory	Optional `String` parameter. Specifies the intermediate directory where tracking logs for this task are stored.
TypeLibraryFile	Optional **String** parameter. Specifies the name of the type library (*.tlb*) file. If you specify this parameter, also specify the **ComponentFileNameMT** task parameter. For more information, see the **/tlb** option in Mt.exe.
UpdateFileHashes	Optional `Boolean` parameter. If `true`, computes the hash value of the files at the path specified by the **UpdateFileHashesSearchPathMT** task parameter, and then updates the value of the **hash** attribute of the **file** element of the manifest by using the computed value. For more information, see the **/hashupdate** option in Mt.exe. Also see the **UpdateFileHashesSearchPath** parameter in this table.
UpdateFileHashesSearchPath	Optional `String` parameter. Specifies the search path to use when the file hashes are updated. Use this parameter with the **UpdateFileHashesMT** task parameter. For more information, see the **UpdateFileHashes** parameter in this table.
VerboseOutput	Optional `Boolean` parameter. If `true`, displays verbose debugging information. For more information, see the **/verbose** option in Mt.exe.

See also

- Task reference

MultiToolTask task

10/21/2019 • 2 minutes to read • Edit Online

No description.

Parameters

The following table describes the parameters of the **MultiToolTask** task.

PARAMETER	DESCRIPTION
EnvironmentVariablesToSet	Optional **string[]** parameter.
SemaphoreProcCount	Optional **string** parameter.
SchedulerFunction	Optional **string** parameter.
SchedulerVerbose	Optional **bool** parameter.
Sources	Required **ITaskItem[]** parameter.
TaskAssemblyName	Optional **string** parameter.
TaskName	Required **string** parameter.
TrackerLogDirectory	Required **string** parameter.

See also

Task reference

ParallelCustomBuild task

10/21/2019 • 2 minutes to read • Edit Online

Run parallel instances of the CustomBuild task.

Parameters

The following table describes the parameters of the **ParallelCustomBuild** task.

PARAMETER	DESCRIPTION
BreakOnFirstFailure	Optional **bool** parameter.
MaxItemsInBatch	Optional **int** parameter.
MaxProcesses	Optional **int** parameter.
Sources	Required **ITaskItem[]** parameter.

See also

Task reference

RC task

10/24/2019 • 2 minutes to read • Edit Online

Wraps the Microsoft Windows Resource Compiler tool, *rc.exe*. The **RC** task compiles resources, such as cursors, icons, bitmaps, dialog boxes, and fonts, into a resource (*.res*) file. For more information, see Resource Compiler.

Parameters

The following table describes the parameters of the RC task. Most task parameters, and a few sets of parameters, correspond to a command-line option.

PARAMETER	DESCRIPTION
AdditionalIncludeDirectories	Optional **String[]** parameter. Adds a directory to the list of directories that are searched for include files. For more information, see the **/I** option in Using RC (the RC command line).
AdditionalOptions	Optional **String** parameter. A list of command-line options; for example, /<option1> /<option2> /<option#>. Use this parameter to specify command-line options that are not represented by any other **RC** task parameter. For more information, see the options in Using RC (the RC command line).
Culture	Optional **String** parameter. Specifies a locale ID that represents the culture used in the resources. For more information, see the **/l** option in Using RC (the RC command line).
IgnoreStandardIncludePath	Optional **Boolean** parameter. If `true`, prevents the resource compiler from checking the INCLUDE environment variable when it searches for header files or resource files. For more information, see the **/x** option in Using RC (the RC command line).
NullTerminateStrings	Optional **Boolean** parameter. If `true`, null-terminates all strings in the string table. For more information, see the **/n** option in Using RC (the RC command line).

PARAMETER	DESCRIPTION
PreprocessorDefinitions	Optional **String[]** parameter. Define one or more preprocessor symbols for the resource compiler. Specify a list of macro symbols. For more information, see the **/d** option in Using RC (the RC command line). Also see **UndefinePreprocessorDefinitions** in this table.
ResourceOutputFileName	Optional **String** parameter. Specifies the name of the resource file. Specify a resource file name. For more information, see the **/fo** option in Using RC (the RC command line).
ShowProgress	Optional **Boolean** parameter. If `true`, displays messages that report on the progress of the compiler. For more information, see the **/v** option in Using RC (the RC command line).
Source	Required `ITaskItem[]` parameter. Defines an array of MSBuild source file items that can be consumed and emitted by tasks.
SuppressStartupBanner	Optional **Boolean** parameter. If `true`, prevents the display of the copyright and version number message when the task starts. For more information, type the **/?** command-line option and then see the **/nologo** option.
TrackerLogDirectory	Optional **String** parameter. Specifies the tracker log directory.
UndefinePreprocessorDefinitions	Undefine a preprocessor symbol. For more information, see the **/u** option in Using RC (the RC command line). Also see **PreprocessorDefinitions** in this table.

See also

- Task reference

SetEnv task

10/21/2019 • 2 minutes to read • Edit Online

Sets or deletes the value of a specified environment variable.

Parameters

The following table describes the parameters of the **SetEnv** task.

PARAMETER	DESCRIPTION
Name	Required **String** parameter. The name of an environment variable.
OutputEnvironmentVariable	Optional **String** output parameter. Contains the value that is assigned to the environment variable that is specified by the **Name** parameter.
Prefix	Mandatory `Boolean` parameter. If `true`, concatenates the value of the **Value** parameter before the value of the environment variable that is specified by the **Name** parameter, and then assigns the result to the environment variable. If `false`, assigns only the value of the **Value** parameter to the environment variable.
Target	Optional **String** parameter. Specifies the location where an environment variable is stored. Specify "User" or "Machine". For more information, see EnvironmentVariableTarget Enumeration.
Value	Optional **String** parameter. The value assigned to the environment variable that is specified by the **Name** parameter. If **Value** is empty and the variable exists, the variable is deleted. If the variable does not exist, no error occurs even though the operation cannot be performed. For more information, see Environment::SetEnvironmentVariable Method.

See also

- Task reference

TrackedVCToolTask base class

3/27/2019 • 2 minutes to read • Edit Online

Many tasks ultimately inherit from the Task class and ToolTask class. This class adds several parameters to the tasks that derive from VCToolTask. These parameters are listed in this document.

Parameters

The following table describes the parameters of the **TrackedVCToolTask** base class.

PARAMETER	DESCRIPTION
DeleteOutputOnExecute	Optional **bool** parameter.
EnableExecuteTool	Optional **bool** parameter.
ExcludedInputPaths	Optional **ITaskItem[]** parameter.
MinimalRebuildFromTracking	Optional **bool** parameter.
PathOverride	Optional **string** parameter.
PostBuildTrackingCleanup	Optional **bool** parameter.
RootSource	Optional **string** parameter.
SkippedExecution	Optional **bool** output parameter.
SourcesCompiled	Optional **ITaskItem[]** output parameter.
TLogCommandFile	Optional **ITaskItem** parameter.
TLogReadFiles	Optional **ITaskItem[]** parameter.
TLogWriteFiles	Optional **ITaskItem[]** parameter.
ToolArchitecture	Optional **string** parameter.
TrackCommandLines	Optional **bool** parameter.
TrackFileAccess	Optional **bool** parameter.
TrackedInputFilesToIgnore	Optional **ITaskItem[]** parameter.
TrackedOutputFilesToIgnore	Optional **ITaskItem[]** parameter.
TrackerFrameworkPath	Optional **string** parameter.
TrackerSdkPath	Optional **string** parameter.

See also

Task reference
Tasks

VCMessage task

10/21/2019 • 2 minutes to read • Edit Online

Logs warning and error messages during a build.

Remarks

This task helps implement MSBuild for C++ projects and is not intended to be called by the user. For more information, see TaskLoggingHelper.

Parameters

The following table describes the parameters of the **VCMessage** task.

PARAMETER	DESCRIPTION
Arguments	Optional **String** parameter. A semicolon-delimited list of messages to display.
Code	Required **String** parameter. An error number that qualifies the message.
Type	Optional **String** parameter. Specifies the kind of message to emit. Specify either "Warning" to emit a warning message, or "Error" to emit an error message.

See also

- Task reference

VCToolTask base class

3/27/2019 • 2 minutes to read • Edit Online

Many tasks ultimately inherit from the Task class and ToolTask class. This class adds several parameters to the tasks that derive from them. These parameters are listed in this document.

Parameters

The following table describes the parameters of the **VCToolTask** base class.

PARAMETER	DESCRIPTION
ActiveToolSwitchesValues	Optional **Dictionary<string, ToolSwitch>** parameter.
AdditionalOptions	Optional **string** parameter.
EffectiveWorkingDirectory	Optional **string** parameter.
EnableErrorListRegex	Optional **bool** parameter. Default is `true`.
ErrorListRegex	Optional **ITaskItem[]** parameter.
ErrorListListExclusion	Optional **ITaskItem[]** parameter.
GenerateCommandLine	Optional **string** parameter. Uses values **CommandLineFormat** *format* [default = CommandLineFormat.ForBuildLog] and **EscapeFormat** *escapeFormat* [default = EscapeFormat.Default].
GenerateCommandLineExceptSwitches	Optional **string** parameter. Uses values **string[]** *switchesToRemove*, **CommandLineFormat** *format* [default = CommandLineFormat.ForBuildLog], and **EscapeFormat** *escapeFormat* [default = EscapeFormat.Default].

See also

Task reference
Tasks

XDCMake task

10/21/2019 • 2 minutes to read • Edit Online

Wraps the XML Documentation tool (*xdcmake.exe*), which merges XML document comment (*.xdc*) files into an *.xml* file.

An *.xdc* file is created when you provide documentation comments in your C++ source code and compile by using the /doc compiler option. For more information, see XDCMake reference, XML Document Generator Tool property pages, and command-line help option (**/?**) for *xdcmake.exe*.

Remarks

By default, the *xdcmake.exe* tool supports a few command-line options. Additional options are supported when you specify the **/old** command-line option.

Parameters

The following table describes the parameters of the **XDCMake** task.

PARAMETER	DESCRIPTION
AdditionalDocumentFile	Optional **String[]** parameter. Specifies one or more additional *.xdc* files to merge. For more information, see the **Additional Document Files** description in XML Document Generator Tool property pages. Also see the **/old** and **/Fs** command-line options for *xdcmake.exe*.
AdditionalOptions	Optional **String** parameter. A list of options as specified on the command line. For example, /<option1> /<option2> /<option#>. Use this parameter to specify options that are not represented by any other **XDCMake** task parameter. For more information, see XDCMake reference, XML Document Generator Tool property pages, and command-line help (**/?**) for *xdcmake.exe*.
DocumentLibraryDependencies	Optional **Boolean** parameter. If `true` and the current project has a dependency on a static library (*.lib*) project in the solution, the *.xdc* files for that library project are included in the *.xml* file output for the current project. For more information, see the **Document Library Dependencies** description in XML Document Generator Tool property pages.

PARAMETER	DESCRIPTION
OutputFile	Optional **String** parameter. Overrides the default output file name. The default name is derived from the name of the first *.xdc* file that is processed. For more information, see the **/out:<filename>** option in XDCMake reference. Also see the **/old** and **/Fo** command-line options for *xdcmake.exe*.
ProjectName	Optional **String** parameter. The name of the current project.
SlashOld	Optional **Boolean** parameter. If `true`, enables additional *xdcmake.exe* options. For more information, see the **/old** command-line option for *xdcmake.exe*.
Sources	Required `ITaskItem[]` parameter. Defines an array of MSBuild source file items that can be consumed and emitted by tasks.
SuppressStartupBanner	Optional **Boolean** parameter. If `true`, prevents the display of the copyright and version number message when the task starts. For more information, see the **/nologo** option in XDCMake reference.
TrackerLogDirectory	Optional **String** parameter. Specifies the directory for the tracker log.

See also

- Task reference

XSD task

10/21/2019 • 2 minutes to read • Edit Online

Wraps the XML Schema Definition tool (*xsd.exe*), which generates schema or class files from a source.

> **NOTE**
> Starting in Visual Studio 2017, C++ project support for *xsd.exe* is deprecated. You can still use the **Microsoft.VisualC.CppCodeProvider** APIs by manually adding *CppCodeProvider.dll* to the GAC.

Parameters

The following table describes the parameters of the **XSD** task.

- **AdditionalOptions**

  Optional **String** parameter.

  A list of options as specified on the command line. For example, /<option1> /<option2> /<option#>. Use this parameter to specify options that are not represented by any other **XSD** task parameter.

- **GenerateFromSchema**

  Optional **String** parameter.

  Specifies the types that are generated from the specified schema.

  Specify one of the following values, each of which corresponds to an XSD option.

  - **classes** - **/classes**
  - **dataset** - **/dataset**

- **Language**

  Optional **String** parameter.

  Specifies the programming language to use for the generated code.

  Choose from **CS** (C#, which is the default), **VB** (Visual Basic), or **JS** (JScript). You can also specify a fully qualified name for a class that implements `System.CodeDom.Compiler.CodeDomProvider Class`.

- **Namespace**

  Optional **String** parameter.

  Specifies the runtime namespace for the generated types.

- **Sources**

  Required `ITaskItem[]` parameter.

  Defines an array of MSBuild source file items that can be consumed and emitted by tasks.

- **SuppressStartupBanner**

  Optional **Boolean** parameter.

If `true`, prevents the display of the copyright and version number message when the task starts.

- **TrackerLogDirectory**

  Optional **String** parameter.

  Specifies the directory for the tracker log.

See also

- Task reference

Task base class

2/21/2019 • 2 minutes to read • Edit Online

Many tasks ultimately inherit from the Task class. This class adds several parameters to the tasks that derive from them. These parameters are listed in this document.

Parameters

The following table describes the parameters of this base class.

PARAMETER	DESCRIPTION
BuildEngine	Optional IBuildEngine parameter. Specifies the build engine interface available to tasks. The build engine automatically sets this parameter to allow tasks to call back into it.
BuildEngine2	Optional IBuildEngine2 parameter. Specifies the build engine interface available to tasks. The build engine automatically sets this parameter to allow tasks to call back into it. This is a convenience property so that task authors inheriting from this class do not have to cast the value from `IBuildEngine` to `IBuildEngine2`.
BuildEngine3	Optional IBuildEngine3 parameter. Specifies the build engine interface provided by the host.
HostObject	Optional ITaskHost parameter. Specifies the host object instance (can be null). The build engine sets this property if the host IDE has associated a host object with this particular task.
Log	Optional TaskLoggingHelper read-only parameter. The logging helper object..

See also

- Task reference
- Tasks

TaskExtension base class

2/21/2019 • 2 minutes to read • Edit Online

Many tasks inherit from the TaskExtension class, which itself inherits from the Task class. This inheritance chain adds several parameters to the tasks that derive from them. These parameters are listed in this document.

Parameters

The following table describes the parameters of the base classes.

PARAMETER	DESCRIPTION
BuildEngine	Optional IBuildEngine parameter. Specifies the build engine interface available to tasks. The build engine automatically sets this parameter to allow tasks to call back into it.
BuildEngine2	Optional IBuildEngine2 parameter. Specifies the build engine interface available to tasks. The build engine automatically sets this parameter to allow tasks to call back into it. This is a convenience property so that task authors inheriting from this class do not have to cast the value from `IBuildEngine` to `IBuildEngine2`.
BuildEngine3	Optional IBuildEngine3 parameter. Specifies the build engine interface provided by the host.
HostObject	Optional ITaskHost parameter. Specifies the host object instance (can be null). The build engine sets this property if the host IDE has associated a host object with this particular task.
Log	Optional TaskLoggingHelper read-only parameter. Gets a `TaskLoggingHelperExtension` object that contains task logging methods.

See also

- Task reference
- Tasks

ToolTaskExtension base class

10/1/2019 • 2 minutes to read • Edit Online

Many tasks inherit from the ToolTaskExtension class, which inherits from the ToolTask class, which itself inherits from the Task class. This inheritance chain adds several parameters to the tasks that derive from them. These parameters are listed in this document.

Parameters

The following table describes the parameters of the base classes.

PARAMETER	DESCRIPTION
BuildEngine	Optional IBuildEngine parameter. Specifies the build engine interface available to tasks. The build engine automatically sets this parameter to allow tasks to call back into it.
BuildEngine2	Optional IBuildEngine2 parameter. Specifies the build engine interface available to tasks. The build engine automatically sets this parameter to allow tasks to call back into it. This is a convenience property so that task authors inheriting from this class do not have to cast the value from `IBuildEngine` to `IBuildEngine2`.
BuildEngine3	Optional IBuildEngine3 parameter. Specifies the build engine interface provided by the host.
EchoOff	Optional `bool` parameter. When set to `true`, this task passes **/Q** to the *cmd.exe* command line such that the command line does not get copied to stdout.
EnvironmentVariables	Optional `String` array parameter. Array of pairs of environment variables, separated by equal signs. These variables are passed to the spawned executable in addition to, or selectively overriding, the regular environment block.
ExitCode	Optional `Int32` output read-only parameter. Specifies the exit code that is provided by the executed command. If the task logged any errors, but the process had an exit code of 0 (success), this is set to -1.

PARAMETER	DESCRIPTION
HostObject	Optional ITaskHost parameter. Specifies the host object instance (can be null). The build engine sets this property if the host IDE has associated a host object with this particular task.
Log	Optional TaskLoggingHelper read-only parameter. Gets an instance of a TaskLoggingHelperExtension class that contains task logging methods.
LogStandardErrorAsError	Option `bool` parameter. If `true`, all messages received on the standard error stream are logged as errors.
StandardErrorImportance	Optional `String` parameter. Importance with which to log text from the standard out stream.
StandardOutputImportance	Optional `String` parameter. Importance with which to log text from the standard out stream.
Timeout	Virtual optional `Int32` parameter. Specifies the amount of time, in milliseconds, after which the task executable is terminated. The default value is `Int.MaxValue`, indicating that there is no time out period. Time-out is in milliseconds.
ToolExe	Virtual optional `string` parameter. Projects may implement this to override a ToolName. Tasks may override this to preserve the ToolName.
ToolPath	Optional `string` parameter. Specifies the location from where the task loads the underlying executable file. If this parameter is not specified, the task uses the SDK installation path that corresponds to the version of the framework that is running MSBuild.
UseCommandProcessor	Optional `bool` parameter. When set to `true`, this task creates a batch file for the command line and executes it by using the command-processor instead of executing the command directly.
YieldDuringToolExecution	Optional `bool` parameter. When set to `true`, this task yields the node when its task is executing.

See also

- Task reference
- Tasks

Diagnosing task failures

10/1/2019 • 2 minutes to read • Edit Online

`MSB6006` is emitted when a ToolTask–derived class runs a tool process that returns a nonzero exit code if the task did not log a more specific error.

Identifying the failing task

When you encounter a task error, the first step is to identify the task that is failing.

The text of the error specifies the tool name (either a friendly name provided by the task's implementation of ToolName or the name of the executable) and the numeric exit code. For example, in

```
error MSB6006: "custom tool" exited with code 1.
```

The tool name is `custom tool` and the exit code is `1`.

Command-line builds

If the build was configured to include a summary (the default), the summary will look like this:

```
Build FAILED.

"S:\MSB6006_demo\MSB6006_demo.csproj" (default target) (1) ->
(InvokeToolTask target) ->
  S:\MSB6006_demo\MSB6006_demo.csproj(19,5): error MSB6006: "custom tool" exited with code 1.
```

This result indicates that the error occurred in a task defined on line 19 of the file `S:\MSB6006_demo\MSB6006_demo.csproj`, in a target named `InvokeToolTask`, in the project `S:\MSB6006_demo\MSB6006_demo.csproj`.

In Visual Studio

The same information is available in the Visual Studio error list in the columns `Project`, `File`, and `Line`.

Finding more failure information

This error is emitted when the task did not log a specific error. The failure to log an error is often because the task is not configured to understand the error format emitted by the tool it calls.

Well-behaved tools generally emit some contextual or error information to their standard output or error stream, and tasks capture and log this information by default. Look in the log entries before the error occurred for additional information. Rerunning the build with a higher log level may be required to preserve this information.

Next steps

Hopefully, the additional context or errors identified in logging reveal the root cause of the problem.

If they do not, you may have to narrow down the potential causes by examining the properties and items that are inputs to the failing task.

AL (Assembly Linker) task

4/16/2019 • 7 minutes to read • Edit Online

The AL task wraps *AL.exe*, a tool that is distributed with the Windows Software Development Kit (SDK). This Assembly Linker tool is used to create an assembly with a manifest from one or more files that are either modules or resource files. Compilers and development environments might already provide these capabilities, so it is often not necessary to use this task directly. The Assembly Linker is most useful to developers needing to create a single assembly from multiple component files, such as those that might be produced from mixed-language development. This task does not combine the modules into a single assembly file; the individual modules must still be distributed and available in order for the resulting assembly to load correctly. For more information on *AL.exe*, see Al.exe (Assembly Linker).

Parameters

The following table describes the parameters of the `AL` task.

PARAMETER	DESCRIPTION
`AlgorithmID`	Optional `String` parameter. Specifies an algorithm to hash all files in a multifile assembly except the file that contains the assembly manifest. For more information, see the documentation for the `/algid` option in Al.exe (Assembly Linker).
`BaseAddress`	Optional `String` parameter. Specifies the address at which a DLL will be loaded on the user's computer at run time. Applications load faster if you specify the base address of the DLLs, rather than letting the operating system relocate the DLLs in the process space. This parameter corresponds to the /baseaddress.
`CompanyName`	Optional `String` parameter. Specifies a string for the `Company` field in the assembly. For more information, see the documentation for the `/comp[any]` option in Al.exe (Assembly Linker).
`Configuration`	Optional `String` parameter. Specifies a string for the `Configuration` field in the assembly. For more information, see the documentation for the `/config[uration]` option in Al.exe (Assembly Linker).
`Copyright`	Optional `String` parameter. Specifies a string for the `Copyright` field in the assembly. For more information, see the documentation for the `/copy[right]` option in Al.exe (Assembly Linker).

PARAMETER	DESCRIPTION
`Culture`	Optional `String` parameter. Specifies the culture string to associate with the assembly. For more information, see the documentation for the `/c[ulture]` option in Al.exe (Assembly Linker).
`DelaySign`	Optional `Boolean` parameter. `true` to place only the public key in the assembly; `false` to fully sign the assembly. For more information, see the documentation for the `/delay[sign]` option in Al.exe (Assembly Linker).
`Description`	Optional `String` parameter. Specifies a string for the `Description` field in the assembly. For more information, see the documentation for the `/descr[iption]` option in Al.exe (Assembly Linker).
`EmbedResources`	Optional ITaskItem `[]` parameter. Embeds the specified resources in the image that contains the assembly manifest. This task copies the contents of the resource file into the image. The items passed in to this parameter may have optional metadata attached to them called `LogicalName` and `Access`. The `LogicalName` metadata is used to specify the internal identifier for the resource. The `Access` metadata can be set to `private` in order to make the resource not visible to other assemblies. For more information, see the documentation for the `/embed[resource]` option in Al.exe (Assembly Linker).
`EvidenceFile`	Optional `String` parameter. Embeds the specified file in the assembly with the resource name of `Security.Evidence`. You cannot use `Security.Evidence` for regular resources. This parameter corresponds to the `/e[vidence]` option in Al.exe (Assembly Linker).
`ExitCode`	Optional `Int32` output read-only parameter. Specifies the exit code provided by the executed command.
`FileVersion`	Optional `String` parameter. Specifies a string for the `File Version` field in the assembly. For more information, see the documentation for the `/fileversion` option in Al.exe (Assembly Linker).
`Flags`	Optional `String` parameter. Specifies a value for the `Flags` field in the assembly. For more information, see the documentation for the `/flags` option in Al.exe (Assembly Linker).

PARAMETER	DESCRIPTION
`GenerateFullPaths`	Optional `Boolean` parameter. Causes the task to use the absolute path for any files that are reported in an error message. This parameter corresponds to the `/fullpaths` option in Al.exe (Assembly Linker).
`KeyContainer`	Optional `String` parameter. Specifies a container that holds a key pair. This will sign the assembly (give it a strong name) by inserting a public key into the assembly manifest. The task will then sign the final assembly with the private key. For more information, see the documentation for the `/keyn[ame]` option in Al.exe (Assembly Linker).
`KeyFile`	Optional `String` parameter. Specifies a file that contains a key pair or just a public key to sign an assembly. The compiler inserts the public key in the assembly manifest and then signs the final assembly with the private key. For more information, see the documentation for the `/keyf[ile]` option in Al.exe (Assembly Linker).
`LinkResources`	Optional ITaskItem `[]` parameter. Links the specified resource files to an assembly. The resource becomes part of the assembly, but the file is not copied. The items passed in to this parameter may have optional metadata attached to them called `LogicalName`, `Target`, and `Access`. The `LogicalName` metadata is used to specify the internal identifier for the resource. The `Target` metadata can specify the path and filename to which the task copies the file, after which it compiles this new file into the assembly. The `Access` metadata can be set to `private` in order to make the resource not visible to other assemblies. For more information, see the documentation for the `/link[resource]` option in Al.exe (Assembly Linker).
`MainEntryPoint`	Optional `String` parameter. Specifies the fully qualified name (*class.method*) of the method to use as an entry point when converting a module to an executable file. This parameter corresponds to the `/main` option in Al.exe (Assembly Linker).
`OutputAssembly`	Required ITaskItem output parameter. Specifies the name of the file generated by this task. This parameter corresponds to the `/out` option in Al.exe (Assembly Linker).
`Platform`	Optional `String` parameter. Limits which platform this code can run on; must be one of `x86`, `Itanium`, `x64`, or `anycpu`. The default is `anycpu`. This parameter corresponds to the `/platform` option in Al.exe (Assembly Linker).

PARAMETER	DESCRIPTION
`ProductName`	Optional `String` parameter. Specifies a string for the `Product` field in the assembly. For more information, see the documentation for the `/prod[uct]` option in Al.exe (Assembly Linker).
`ProductVersion`	Optional `String` parameter. Specifies a string for the `ProductVersion` field in the assembly. For more information, see the documentation for the `/productv[ersion]` option in Al.exe (Assembly Linker).
`ResponseFiles`	Optional `String[]` parameter. Specifies the response files that contain additional options to pass through to the Assembly Linker.
`SdkToolsPath`	Optional `String` parameter. Specifies the path to the SDK tools, such as resgen.exe.
`SourceModules`	Optional ITaskItem `[]` parameter. One or more modules to be compiled into an assembly. The modules will be listed in the manifest of the resulting assembly, and will still need to distributed and available in order for the assembly to load. The items passed into this parameter may have additional metadata called `Target`, which specifies the path and filename to which the task copies the file, after which it compiles this new file into the assembly. For more information, see the documentation for Al.exe (Assembly Linker). This parameter corresponds to the list of modules passed into *Al.exe* without a specific switch.
`TargetType`	Optional `String` parameter. Specifies the file format of the output file: `library` (code library), `exe` (console application), or `win` (Windows-based application). The default is `library`. This parameter corresponds to the `/t[arget]` option in Al.exe (Assembly Linker).
`TemplateFile`	Optional `String` parameter. Specifies the assembly from which to inherit all assembly metadata, except the culture field. The specified assembly must have a strong name. An assembly that you create with the `TemplateFile` parameter will be a satellite assembly. This parameter corresponds to the `/template` option in Al.exe (Assembly Linker).

PARAMETER	DESCRIPTION
`Timeout`	Optional `Int32` parameter. Specifies the amount of time, in milliseconds, after which the task executable is terminated. The default value is `Int.MaxValue`, indicating that there is no time out period.
`Title`	Optional `String` parameter. Specifies a string for the `Title` field in the assembly. For more information, see the documentation for the `/title` option in Al.exe (Assembly Linker).
`ToolPath`	Optional `String` parameter. Specifies the location from where the task will load the underlying executable file (Al.exe). If this parameter is not specified, the task uses the SDK installation path corresponding to the version of the framework that is running MSBuild.
`Trademark`	Optional `String` parameter. Specifies a string for the `Trademark` field in the assembly. For more information, see the documentation for the `/trade[mark]` option in Al.exe (Assembly Linker).
`Version`	Optional `String` parameter. Specifies the version information for this assembly. The format of the string is *major.minor.build.revision*. The default value is 0. For more information, see the documentation for the `/v[ersion]` option in Al.exe (Assembly Linker).
`Win32Icon`	Optional `String` parameter. Inserts an *.ico* file in the assembly. The *.ico* file gives the output file the desired appearance in File Explorer. This parameter corresponds to the `/win32icon` option in Al.exe (Assembly Linker).
`Win32Resource`	Optional `String` parameter. Inserts a Win32 resource (*.res* file) in the output file. For more information, see the documentation for the `/win32res` option in Al.exe (Assembly Linker).

Remarks

In addition to the parameters listed above, this task inherits parameters from the ToolTaskExtension class, which itself inherits from the ToolTask class. For a list of these additional parameters and their descriptions, see ToolTaskExtension base class.

Example

The following example creates an assembly with the specified options.

```
<AL
    EmbedResources="@(EmbeddedResource)"
    Culture="%(EmbeddedResource.Culture)"
    TemplateFile="@(IntermediateAssembly)"
    KeyContainer="$(KeyContainerName)"
    KeyFile="$(KeyOriginatorFile)"
    DelaySign="$(DelaySign)"

    OutputAssembly=
       "%(EmbeddedResource.Culture)\$(TargetName).resources.dll">

    <Output TaskParameter="OutputAssembly"
        ItemName="SatelliteAssemblies"/>
</AL>
```

See also

- Task reference
- Tasks

AspNetCompiler task

2/21/2019 • 2 minutes to read • Edit Online

The `AspNetCompiler` task wraps *aspnet_compiler.exe*, a utility to precompile ASP.NET applications.

Task parameters

The following table describes the parameters of the `AspNetCompiler` task.

PARAMETER	DESCRIPTION
`AllowPartiallyTrustedCallers`	Optional `Boolean` parameter. If this parameter is `true`,, the strong-name assembly will allow partially trusted callers.
`Clean`	Optional `Boolean` parameter If this parameter is `true`, the precompiled application will be built clean. Any previously compiled components will be recompiled. The default value is `false`. This parameter corresponds to the **-c** switch on *aspnet_compiler.exe*.
`Debug`	Optional `Boolean` parameter. If this parameter is `true`, debug information (.PDB file) is emitted during compilation. The default value is `false`. This parameter corresponds to the **-d** switch on *aspnet_compiler.exe*.
`DelaySign`	Optional `Boolean` parameter. If this parameter is `true`, the assembly is not fully signed when created.
`FixedNames`	Optional `Boolean` parameter. If this parameter is `true`, the compiled assemblies will be given fixed names..
`Force`	Optional `Boolean` parameter If this parameter is `true`, the task will overwrite the target directory if it already exists. Existing contents are lost. The default value is `false`. This parameter corresponds to the **-f** switch on *aspnet_compiler.exe*.
`KeyContainer`	Optional `String` parameter. Specifies a strong name key container.

PARAMETER	DESCRIPTION
`KeyFile`	Optional `String` parameter. Specifies the physical path to the strong name key file..
`MetabasePath`	Optional `String` parameter. Specifies the full IIS metabase path of the application. This parameter cannot be combined with the `VirtualPath` or `PhysicalPath` parameters. This parameter corresponds to the **-m** switch on *aspnet_compiler.exe*.
`PhysicalPath`	Optional `String` parameter. Specifies the physical path of the application to be compiled. If this parameter is missing, the IIS metabase is used to locate the application. This parameter corresponds to the **-p** switch on *aspnet_compiler.exe*.
`TargetFrameworkMoniker`	Optional `String` parameter. Specifies the TargetFrameworkMoniker indicating which .NET Framework version of *aspnet_compiler.exe* should be used. Only accepts .NET Framework monikers.
`TargetPath`	Optional `String` parameter. Specifies the physical path to which the application is compiled. If not specified, the application is precompiled in-place.
`Updateable`	Optional `Boolean` parameter. If this parameter is `true`, the precompiled application will be updateable. The default value is `false`. This parameter corresponds to the **-u** switch on *aspnet_compiler.exe*.
`VirtualPath`	Optional `String` parameter. The virtual path of the application to be compiled. If `PhysicalPath` specified, the physical path is used to locate the application. Otherwise, the IIS metabase is used, and the application is assumed to be in the default site. This parameter corresponds to the **-v** switch on *aspnet_compiler.exe*.

Remarks

In addition to the parameters listed above, this task inherits parameters from the ToolTaskExtension class, which itself inherits from the ToolTask class. For a list of these additional parameters and their descriptions, see ToolTaskExtension base class.

Example

The following code example uses the `AspNetCompiler` task to precompile an ASP.NET application.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <Target Name="PrecompileWeb">
        <AspNetCompiler
            VirtualPath="/MyWebSite"
            PhysicalPath="c:\inetpub\wwwroot\MyWebSite\"
            TargetPath="c:\precompiledweb\MyWebSite\"
            Force="true"
            Debug="true"
        />
    </Target>
</Project>
```

See also

- Tasks
- Task reference

AssignCulture task

2/21/2019 • 2 minutes to read • Edit Online

This task accepts a list of items that may contain a valid .NET culture identifier string as part of the file name, and produces items that have a metadata named `Culture` containing the corresponding culture identifier. For example, the file name *Form1.fr-fr.resx* has an embedded culture identifier "fr-fr", so this task will produce an item that has the same filename with the metadata `Culture` equal to `fr-fr`. The task also produces a list of filenames with the culture removed from the filename.

Task parameters

The following table describes the parameters of the `AssignCulture` task.

PARAMETER	DESCRIPTION
`AssignedFiles`	Optional ITaskItem `[]` output parameter. Contains the list of items received in the `Files` parameter, with a `Culture` metadata entry added to each item. If the incoming item from the `Files` parameter already contains a `Culture` metadata entry, the original metadata entry is used. The task only assigns a `Culture` metadata entry if the file name contains a valid culture identifier. The culture identifier must be between the last two dots in the filename.
`AssignedFilesWithCulture`	Optional ITaskItem `[]` output parameter. Contains the subset of the items from the `AssignedFiles` parameter that have a `Culture` metadata entry.
`AssignedFilesWithNoCulture`	Optional ITaskItem `[]` output parameter. Contains the subset of the items from the `AssignedFiles` parameter that do not have a `Culture` metadata entry.
`CultureNeutralAssignedFiles`	Optional ITaskItem `[]` output parameter. Contains the same list of items that is produced in the `AssignedFiles` parameter, except with the culture removed from the file name. The task only removes the culture from the file name if it is a valid culture identifier.
`Files`	Required ITaskItem `[]` parameter. Specifies the list of files with embedded culture names to assign a culture to.

Remarks

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following example executes the `AssignCulture` task with the `ResourceFiles` item collection.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <ItemGroup>
        <ResourceFiles Include="MyResource1.fr.resx"/>
        <ResourceFiles Include="MyResource2.XX.resx"/>
    </ItemGroup>

    <Target Name="Culture">
        <AssignCulture
            Files="@(ResourceFiles)"
            <Output TaskParameter="AssignedFiles"
                ItemName="OutAssignedFiles"/>
            <Output TaskParameter="AssignedFilesWithCulture"
                ItemName="OutAssignedFilesWithCulture"/>
            <Output TaskParameter="AssignedFilesWithNoCulture"
                ItemName="OutAssignedFilesWithNoCulture"/>
            <Output TaskParameter="CultureNeutralAssignedFiles"
                ItemName="OutCultureNeutralAssignedFiles"/>
        </AssignCulture>
    </Target>
</Project>
```

The following table describes the value of the output items after task execution. Item metadata is shown in parenthesis after the item.

ITEM COLLECTION	CONTENTS
`OutAssignedFiles`	*MyResource1.fr.resx* (Culture="fr") *MyResource2.XX.resx* (no additional metadata)
`OutAssignedFilesWithCulture`	*MyResource1.fr.resx* (Culture="fr")
`OutAssignedFilesWithNoCulture`	*MyResource2.XX.resx* (no additional metadata)
`OutCultureNeutralAssignedFiles`	*MyResource1.resx* (Culture="fr") *MyResource2.XX.resx* (no additional metadata)

See also

- Tasks
- Task reference

AssignProjectConfiguration task

2/21/2019 • 2 minutes to read • Edit Online

This task accepts a list configuration strings and assigns them to specified projects.

Task parameters

The following table describes the parameters of the `AssignProjectConfiguration` task.

PARAMETER	DESCRIPTION
`SolutionConfigurationContents`	Optional `string` output parameter. Contains an XML string containing a project configuration for each project. The configurations are assigned to the named projects.
`DefaultToVcxPlatformMapping`	Optional `string` output parameter. Contains a semicolon-delimited list of mappings from the platform names used by most types to those used by *.vcxproj* files. For example: `"AnyCPU=Win32;X86=Win32;X64=X64"`
`VcxToDefaultPlatformMapping`	Optional `string` output parameter. Contains a semicolon-delimited list of mappings from *.vcxproj* platform names to the platform names use by most types. For example: `"Win32=AnyCPU;X64=X64"`
`CurrentProjectConfiguration`	Optional `string` output parameter. Contains the configuration for the current project.
`CurrentProjectPlatform`	Optional `string` output parameter. Contains the platform for the current project.
`OnlyReferenceAndBuildProjectsEnabledInSolutionConfiguration`	Optional `bool` output parameter. Contains a flag indicating that references should be built even if they were disabled in the project configuration.

PARAMETER	DESCRIPTION
`ShouldUnsetParentConfigurationAndPlatform`	Optional `bool` output parameter. Contains a flag indicating if the parent configuration and platform should be unset.
`OutputType`	Optional `string` output parameter. Contains the output type for the project.
`ResolveConfigurationPlatformUsingMappings`	Optional `bool` output parameter. Contains a flag indicating if the build should use the default mappings to resolve the configuration and platform of the passed in project references.
`AssignedProjects`	Optional ITaskItem `[]` output parameter. Contains the list of resolved reference paths.
`UnassignedProjects`	Optional ITaskItem `[]` output parameter. Contains the list of project reference items that could not be resolved using the pre-resolved list of outputs.

Remarks

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

See also

- Tasks
- Task reference

AssignTargetPath task

2/21/2019 • 2 minutes to read • Edit Online

This task accepts a list of files and adds `<TargetPath>` attributes if they are not already specified.

Task parameters

The following table describes the parameters of the `AssignTargetPath` task.

PARAMETER	DESCRIPTION
`RootFolder`	Optional `string` input parameter. Contains the path to the folder that contains the target links.
`Files`	Optional ITaskItem `[]` input parameter. Contains the incoming list of files.
`AssignedFiles`	Optional ITaskItem `[]` output parameter. Contains the resulting list of files.

Remarks

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following example executes the `AssignTargetPath` task to configure a project.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <Target Name="MyProject">
        <AssignTargetPath
RootFolder="Resources"
            Files="@(ResourceFiles)"
            <Output TaskParameter="AssignedFiles"
                ItemName="OutAssignedFiles"/>
        </AssignTargetPath>
    </Target>
</Project>
```

See also

- Tasks
- Task reference

CallTarget task

4/16/2019 • 2 minutes to read • Edit Online

Invokes the specified targets within the project file.

Task parameters

The following table describes the parameters of the `CallTarget` task.

PARAMETER	DESCRIPTION
`RunEachTargetSeparately`	Optional `Boolean` input parameter. If `true`, the MSBuild engine is called once per target. If `false`, the MSBuild engine is called once to build all targets. The default value is `false`.
`TargetOutputs`	Optional ITaskItem `[]` output parameter. Contains the outputs of all built targets.
`Targets`	Optional `String[]` parameter. Specifies the target or targets to build.
`UseResultsCache`	Optional `Boolean` parameter. If `true`, the cached result is returned if present. **Note** When an MSBuild task is run, its output is cached in a scope (ProjectFileName, GlobalProperties)[TargetNames] as a list of build items.

Remarks

If a target specified in `Targets` fails and `RunEachTargetSeparately` is `true`, the task continues to build the remaining targets.

If you want to build the default targets, use the MSBuild task and set the `Projects` parameter equal to `$(MSBuildProjectFile)`.

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following example calls `TargetA` from inside `CallOtherTargets`.

```
<Project DefaultTargets="CallOtherTargets"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <Target Name="CallOtherTargets">
        <CallTarget Targets="TargetA"/>
    </Target>

    <Target Name="TargetA">
        <Message Text="Building TargetA..." />
    </Target>

</Project>
```

See also

- Task reference
- Targets

CombinePath task

2/21/2019 • 2 minutes to read • Edit Online

Combines the specified paths into a single path.

Task parameters

The following table describes the parameters of the CombinePath task.

PARAMETER	DESCRIPTION
`BasePath`	Required `String` parameter. The base path to combine with the other paths. Can be a relative path, absolute path, or blank.
`Paths`	Required ITaskItem `[]` parameter. A list of individual paths to combine with the BasePath to form the combined path. Paths can be relative or absolute.
`CombinedPaths`	Optional ITaskItem `[]` output parameter. The combined path that is created by this task.

Remarks

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

See also

- Tasks
- Task reference

ConvertToAbsolutePath task

2/21/2019 • 2 minutes to read • Edit Online

Converts a relative path, or reference, into an absolute path.

Task parameters

The following table describes the parameters of the `ConvertToAbsolutePath` task.

PARAMETER	DESCRIPTION
`Paths`	Required ITaskItem `[]` parameter. The list of relative paths to convert to absolute paths.
`AbsolutePaths`	Optional ITaskItem `[]` output parameter. The list of absolute paths for the items that were passed in.

Remarks

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

See also

- Tasks
- Task reference

Copy task

2/21/2019 • 2 minutes to read • Edit Online

Copies files to a new location in the file system.

Parameters

The following table describes the parameters of the `Copy` task.

PARAMETER	DESCRIPTION
`CopiedFiles`	Optional ITaskItem `[]` output parameter. Contains the items that were successfully copied.
`DestinationFiles`	Optional ITaskItem `[]` parameter. Specifies the list of files to copy the source files to. This list is expected to be a one-to-one mapping with the list specified in the `SourceFiles` parameter. That is, the first file specified in `SourceFiles` will be copied to the first location specified in `DestinationFiles`, and so forth.
`DestinationFolder`	Optional ITaskItem parameter. Specifies the directory to which you want to copy the files. This must be a directory, not a file. If the directory does not exist, it is created automatically.
`OverwriteReadOnlyFiles`	Optional `Boolean` parameter. Overwrite files even if they are marked as read only files
`Retries`	Optional `Int32` parameter. Specifies how many times to attempt to copy, if all previous attempts have failed. Defaults to zero. **Note:** The use of retries can mask a synchronization problem in your build process.
`RetryDelayMilliseconds`	Optional `Int32` parameter. Specifies the delay between any necessary retries. Defaults to the RetryDelayMillisecondsDefault argument, which is passed to the CopyTask constructor.

PARAMETER	DESCRIPTION
`SkipUnchangedFiles`	Optional `Boolean` parameter. If `true`, skips the copying of files that are unchanged between the source and destination. The `Copy` task considers files to be unchanged if they have the same size and the same last modified time. **Note:** If you set this parameter to `true`, you should not use dependency analysis on the containing target, because that only runs the task if the last-modified times of the source files are newer than the last-modified times of the destination files.
`SourceFiles`	Required ITaskItem `[]` parameter. Specifies the files to copy.
`UseHardlinksIfPossible`	Optional `Boolean` parameter. If `true`, creates Hard Links for the copied files instead of copying the files.

Warnings

Warnings are logged, including:

- `Copy.DestinationIsDirectory`
- `Copy.SourceIsDirectory`
- `Copy.SourceFileNotFound`
- `Copy.CreatesDirectory`
- `Copy.HardLinkComment`
- `Copy.RetryingAsFileCopy`
- `Copy.FileComment`
- `Copy.RemovingReadOnlyAttribute`

Remarks

Either the `DestinationFolder` or the `DestinationFiles` parameter must be specified, but not both. If both are specified, the task fails and an error is logged.

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following example copies the items in the `MySourceFiles` item collection into the folder *c:\MyProject\Destination*.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <ItemGroup>
        <MySourceFiles Include="a.cs;b.cs;c.cs"/>
    </ItemGroup>

    <Target Name="CopyFiles">
        <Copy
            SourceFiles="@(MySourceFiles)"
            DestinationFolder="c:\MyProject\Destination"
        />
    </Target>

</Project>
```

Example

The following example demonstrates how to do a recursive copy. This project copies all of the files recursively from *c:\MySourceTree* into *c:\MyDestinationTree*, while maintaining the directory structure.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <ItemGroup>
        <MySourceFiles Include="c:\MySourceTree\**\*.*"/>
    </ItemGroup>

    <Target Name="CopyFiles">
        <Copy
            SourceFiles="@(MySourceFiles)"
            DestinationFiles="@(MySourceFiles->'c:\MyDestinationTree\%(RecursiveDir)%(Filename)%(Extension)')"
        />
    </Target>

</Project>
```

See also

- Tasks
- Task reference

CreateCSharpManifestResourceName task

4/16/2019 • 2 minutes to read • Edit Online

Creates a Visual C#-style manifest name from a given *.resx* file name or other resource.

Parameters

The following table describes the parameters of the CreateCSharpManifestResourceName task.

PARAMETER	DESCRIPTION
`ManifestResourceNames`	ITaskItem `[]` output read-only parameter. The resulting manifest names.
`ResourceFiles`	Required `String` parameter. The name of the resource file from which to create the Visual C# manifest name.
`RootNamespace`	Optional `String` parameter. The root namespace of the resource file, typically taken from the project file. May be `null`.
`PrependCultureAsDirectory`	Optional `Boolean` parameter. If `true`, the culture name is added as a directory name just before the manifest resource name. Default value is `true`.
`ResourceFilesWithManifestResourceNames`	Optional read-only `String` output parameter. Returns the name of the resource file that now includes the manifest resource name.

Remarks

The CreateVisualBasicManifestResourceName task determines the appropriate manifest resource name to assign to a given *.resx* or other resource file. The task provides a logical name to a resource file, and then attaches it to an output parameter as metadata.

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

See also

- Tasks
- Task reference

CreateItem task

4/23/2019 • 2 minutes to read • Edit Online

Populates item collections with the input items. This allows items to be copied from one list to another.

> **NOTE**
> This task is deprecated. Starting with .NET Framework 3.5, item groups may be placed within Target elements. For more information, see Items.

Attributes

The following table describes the parameters of the `CreateItem` task.

PARAMETER	DESCRIPTION
`AdditionalMetadata`	Optional `String` array parameter. Specifies additional metadata to attach to the output items. Specify the metadata name and value for the item with the following syntax: *MetadataName* `=` *MetadataValue* Multiple metadata name/value pairs should be separated with a semicolon. If either the name or the value contains a semicolon or any other special characters, they must be escaped. For more information, see How to: Escape special characters in MSBuild.
`Exclude`	Optional ITaskItem `[]` output parameter. Specifies the items to exclude from the output item collection. This parameter can contain wildcard specifications. For more information, see Items and How to: Exclude files from the build.
`Include`	Required ITaskItem `[]` parameter. Specifies the items to include in the output item collection. This parameter can contain wildcard specifications.
`PreserveExistingMetadata`	Optional `Boolean` parameter. If `True`, only apply the additional metadata if they do not already exist.

Remarks

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following code example creates a new item collection named `MySourceItemsWithMetadata` from the item collection `MySourceItems`. The `CreateItem` task populates the new item collection with the items in the `MySourceItems` item. It then adds an additional metadata entry named `MyMetadata` with a value of `Hello` to each item in the new collection.

After the task is executed, the `MySourceItemsWithMetadata` item collection contains the items *file1.resx* and *file2.resx*, both with metadata entries for `MyMetadata`. The `MySourceItems` item collection is unchanged.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <ItemGroup>
        <MySourceItems Include="file1.resx;file2.resx" />
    </ItemGroup>

    <Target Name="NewItems">
        <CreateItem
            Include="@(MySourceItems)"
            AdditionalMetadata="MyMetadata=Hello">
           <Output
               TaskParameter="Include"
               ItemName="MySourceItemsWithMetadata"/>
        </CreateItem>

    </Target>

</Project>
```

The following table describes the value of the output item after task execution. Item metadata is shown in parenthesis after the item.

ITEM COLLECTION	CONTENTS
`MySourceItemsWithMetadata`	*file1.resx* (`MyMetadata="Hello"`) *file2.resx* (`MyMetadata="Hello"`)

See also

- Task reference
- Tasks

CreateProperty task

2/21/2019 • 2 minutes to read • Edit Online

Populates properties with the values passed in. This allows values to be copied from one property or string to another.

Attributes

The following table describes the parameters of the `CreateProperty` task.

PARAMETER	DESCRIPTION
`Value`	Optional `String` output parameter. Specifies the value to copy to the new property.
`ValueSetByTask`	Optional `String` output parameter. Contains the same value as the `Value` parameter. Use this parameter only when you want to avoid having the output property set by MSBuild when it skips the enclosing target because the outputs are up-to-date.

Remarks

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following example uses the `CreateProperty` task to create the `NewFile` property using the combination of the values of the `SourceFilename` and `SourceFileExtension` property.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <PropertyGroup>
        <SourceFilename>Module1</SourceFilename>
        <SourceFileExtension>vb</SourceFileExtension>
    </PropertyGroup>

    <Target Name="CreateProperties">

        <CreateProperty
            Value="$(SourceFilename).$(SourceFileExtension)">
            <Output
                TaskParameter="Value"
                PropertyName="NewFile" />
        </CreateProperty>

    </Target>

</Project>
```

After running the project, the value of the `NewFile` property is *Module1.vb*.

See also

- Task reference
- Tasks

CreateVisualBasicManifestResourceName task

4/16/2019 • 2 minutes to read • Edit Online

Creates a Visual Basic-style manifest name from a given *.resx* file name or other resource.

Parameters

The following table describes the parameters of the CreateVisualBasicManifestResourceName task.

PARAMETER	DESCRIPTION
`ManifestResourceNames`	ITaskItem `[]` output read-only parameter. The resulting manifest names.
`ResourceFiles`	Required `String` parameter. The name of the resource file from which to create the Visual Basic manifest name.
`RootNamespace`	Optional `String` parameter. The root namespace of the resource file, typically taken from the project file. May be `null`.
`PrependCultureAsDirectory`	Optional `Boolean` parameter. If `true`, the culture name is added as a directory name just before the manifest resource name. Default value is `true`.
`ResourceFilesWithManifestResourceNames`	Optional read-only `String` output parameter. Returns the name of the resource file that now includes the manifest resource name.

Remarks

The CreateVisualBasicManifestResourceName task determines the appropriate manifest resource name to assign to a given *.resx* or other resource file. The task provides a logical name to a resource file, and then attaches it to an output parameter as metadata.

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

See also

- Tasks
- Task reference

Csc task

9/11/2019 • 7 minutes to read • Edit Online

Wraps *csc.exe*, and produces executables (*.exe* files), dynamic-link libraries (*.dll* files), or code modules (*.netmodule* files). For more information about *csc.exe*, see C# compiler options.

Parameters

The following table describes the parameters of the `Csc` task.

PARAMETER	DESCRIPTION
`AdditionalLibPaths`	Optional `String[]` parameter. Specifies additional directories to search for references. For more information, see -lib (C# compiler options).
`AddModules`	Optional `String` parameter. Specifies one or more modules to be part of the assembly. For more information, see -addmodule (C# compiler options).
`AllowUnsafeBlocks`	Optional `Boolean` parameter. If `true`, compiles code that uses the unsafe keyword. For more information, see -unsafe (C# compiler options).
`ApplicationConfiguration`	Optional `String` parameter. Specifies the application configuration file containing the assembly binding settings.
`BaseAddress`	Optional `String` parameter. Specifies the preferred base address at which to load a DLL. The default base address for a DLL is set by the .NET Framework common language runtime. For more information, see -baseaddress (C# compiler options).
`CheckForOverflowUnderflow`	Optional `Boolean` parameter. Specifies whether integer arithmetic that overflows the bounds of the data type causes an exception at run time. For more information, see -checked (C# compiler options).
`CodePage`	Optional `Int32` parameter. Specifies the code page to use for all source code files in the compilation. For more information, see -codepage (C# compiler options).

PARAMETER	DESCRIPTION
`DebugType`	Optional `String` parameter. Specifies the debug type. `DebugType` can be `full` or `pdbonly`. The default is `full`, which enables a debugger to be attached to a running program. Specifying `pdbonly` enables source code debugging when the program is started in the debugger, but only displays assembler when the running program is attached to the debugger. This parameter overrides the `EmitDebugInformation` parameter. For more information, see -debug (C# compiler options).
`DefineConstants`	Optional `String` parameter. Defines preprocessor symbols. For more information, see -define (C# compiler options).
`DelaySign`	Optional `Boolean` parameter. If `true`, specifies that you only want to place the public key in the assembly. If `false`, specifies that you want a fully signed assembly This parameter has no effect unless used with either the `KeyFile` or `KeyContainer` parameter. For more information, see -delaysign (C# compiler options).
`Deterministic`	Optional `Boolean` parameter. If `true`, causes the compiler to output an assembly whose binary content is identical across compilations if inputs are identical. For more information, see -deterministic (C# Compiler options).
`DisabledWarnings`	Optional `String` parameter. Specifies the list of warnings to be disabled. For more information, see -nowarn (C# compiler options).
`DocumentationFile`	Optional `String` parameter. Processes documentation comments to an XML file. For more information, see -doc (C# compiler options).
`EmitDebugInformation`	Optional `Boolean` parameter. If `true`, the task generates debugging information and places it in a program database (.pdb) file. If `false`, the task emits no debug information. Default is `false`. For more information, see -debug (C# compiler options).

PARAMETER	DESCRIPTION
`ErrorReport`	Optional `String` parameter. Provides a convenient way to report a C# internal error to Microsoft. This parameter can have a value of `prompt`, `send`, or `none`. If the parameter is set to `prompt`, you will receive a prompt when an internal compiler error occurs. The prompt lets you send a bug report electronically to Microsoft. If the parameter is set to `send`, a bug report is sent automatically. If the parameter is set to `none`, the error is reported only in the text output of the compiler. Default is `none`. For more information, see -errorreport (C# compiler options).
`FileAlignment`	Optional `Int32` parameter. Specifies the size of sections in the output file. For more information, see -filealign (C# compiler options).
`GenerateFullPaths`	Optional `Boolean` parameter. If `true`, specifies the absolute path to the file in the compiler output. If `false`, specifies the name of the file. Default is `false`. For more information, see -fullpaths (C# compiler options).
`KeyContainer`	Optional `String` parameter. Specifies the name of the cryptographic key container. For more information, see -keycontainer (C# compiler options).
`KeyFile`	Optional `String` parameter. Specifies the file name containing the cryptographic key. For more information, see -keyfile (C# compiler options).
`LangVersion`	Optional `String` parameter. Specifies the version of the language to use. For more information, see -langversion (C# compiler options).
`LinkResources`	Optional ITaskItem `[]` parameter. Creates a link to a .NET Framework resource in the output file; the resource file is not placed in the output file. Items passed into this parameter can have optional metadata entries named `LogicalName` and `Access`. `LogicalName` corresponds to the `identifier` parameter of the `/linkresource` switch, and `Access` corresponds to `accessibility-modifier` parameter. For more information, see -linkresource (C# compiler options).
`MainEntryPoint`	Optional `String` parameter. Specifies the location of the `Main` method. For more information, see -main (C# compiler options).

PARAMETER	DESCRIPTION
`ModuleAssemblyName`	Optional `String` parameter. Specifies the name of the assembly that this module will be a part of.
`NoConfig`	Optional `Boolean` parameter. If `true`, tells the compiler not to compile with the *csc.rsp* file. For more information, see -noconfig (C# compiler options).
`NoLogo`	Optional `Boolean` parameter. If `true`, suppresses display of compiler banner information. For more information, see -nologo (C# compiler options).
`NoStandardLib`	Optional `Boolean` parameter. If `true`, prevents the import of *mscorlib.dll*, which defines the entire System namespace. Use this parameter if you want to define or create your own System namespace and objects. For more information, see -nostdlib (C# compiler options).
`NoWin32Manifest`	Optional `Boolean` parameter. If `true`, do not include the default Win32 manifest.
`Optimize`	Optional `Boolean` parameter. If `true`, enables optimizations. If `false`, disables optimizations. For more information, see -optimize (C# compiler options).
`OutputAssembly`	Optional `String` output parameter. Specifies the name of the output file. For more information, see -out (C# compiler options).
`OutputRefAssembly`	Optional `String` parameter. Specifies the name of the output reference assembly file. For more information, see -refout (C# compiler options).
`PdbFile`	Optional `String` parameter. Specifies the debug information file name. The default name is the output file name with a *.pdb* extension.
`Platform`	Optional `String` parameter. Specifies the processor platform to be targeted by the output file. This parameter can have a value of `x86`, `x64`, or `anycpu`. Default is `anycpu`. For more information, see -platform (C# compiler options).

PARAMETER	DESCRIPTION

`References`	Optional ITaskItem `[]` parameter.
	
	Causes the task to import public type information from the specified items into the current project. For more information, see -reference (C# compiler options).
	
	You can specify a Visual C# reference alias in an MSBuild file by adding the metadata `Aliases` to the original "Reference" item. For example, to set the alias "LS1" in the following Csc command line:
	
	`CSC /r:LS1=MyCodeLibrary.dll /r:LS2=MyCodeLibrary2.dll *.cs`
	
	you would use:
	
	`<Reference Include="MyCodeLibrary"> <Aliases>LS1</Aliases> </Reference>`

`Resources`	Optional ITaskItem `[]` parameter. Embeds a .NET Framework resource into the output file. Items passed into this parameter can have optional metadata entries named `LogicalName` and `Access`. `LogicalName` corresponds to the `identifier` parameter of the `/resource` switch, and `Access` corresponds to `accessibility-modifier` parameter. For more information, see -resource (C# compiler options).
`ResponseFiles`	Optional `String` parameter. Specifies the response file that contains commands for this task. For more information, see @ (Specify response file).
`Sources`	Optional ITaskItem `[]` parameter. Specifies one or more Visual C# source files.
`TargetType`	Optional `String` parameter. Specifies the file format of the output file. This parameter can have a value of `library`, which creates a code library, `exe`, which creates a console application, `module`, which creates a module, or `winexe`, which creates a Windows program. The default value is `library`. For more information, see -target (C# compiler options).
`TreatWarningsAsErrors`	Optional `Boolean` parameter. If `true`, treats all warnings as errors. For more information, see -warnaserror (C# compiler options).
`UseHostCompilerIfAvailable`	Optional `Boolean` parameter. Instructs the task to use the in-process compiler object, if available. Used only by Visual Studio.

PARAMETER	DESCRIPTION
`Utf8Output`	Optional `Boolean` parameter. Logs compiler output using UTF-8 encoding. For more information, see -utf8output (C# compiler options).
`WarningLevel`	Optional `Int32` parameter. Specifies the warning level for the compiler to display. For more information, see -warn (C# compiler options).
`WarningsAsErrors`	Optional `String` parameter. Specifies a list of warnings to treat as errors. For more information, see -warnaserror (C# compiler options). This parameter overrides the `TreatWarningsAsErrors` parameter.
`WarningsNotAsErrors`	Optional `String` parameter. Specifies a list of warnings that are not treated as errors. For more information, see -warnaserror (C# compiler options). This parameter is only useful if the `TreatWarningsAsErrors` parameter is set to `true`.
`Win32Icon`	Optional `String` parameter. Inserts an *.ico* file in the assembly, which gives the output file the desired appearance in **File Explorer**. For more information, see -win32icon (C# compiler options).
`Win32Manifest`	Optional `String` parameter. Specifies the Win32 manifest to be included.
`Win32Resource`	Optional `String` parameter. Inserts a Win32 resource (*.res*) file in the output file. For more information, see -win32res (C# compiler options).

Remarks

In addition to the parameters listed above, this task inherits parameters from the `Microsoft.Build.Tasks.ManagedCompiler` class, which inherits from the ToolTaskExtension class, which itself inherits from the ToolTask class. For a list of these additional parameters and their descriptions, see ToolTaskExtension base class.

Example

The following example uses the `Csc` task to compile an executable from the source files in the `Compile` item collection.

```
<CSC
    Sources="@(Compile)"
    OutputAssembly="$(AppName).exe"
    EmitDebugInformation="true" />
```

See also

- Task reference
- Tasks

Delete task

2/21/2019 • 2 minutes to read • Edit Online

Deletes the specified files.

Parameters

The following table describes the parameters of the `Delete` task.

PARAMETER	DESCRIPTION
`DeletedFiles`	Optional ITaskItem `[]` output parameter. Specifies the files that were successfully deleted.
`Files`	Required ITaskItem `[]` parameter. Specifies the files to delete.
`TreatErrorsAsWarnings`	Optional `Boolean` parameter If `true`, errors are logged as warnings. The default value is `false`.

Remarks

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following example deletes the file *MyApp.pdb*.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <PropertyGroup>
        <AppName>MyApp</AppName>
    </PropertyGroup>

    <Target Name="DeleteFiles">
        <Delete Files="$(AppName).pdb" />
    </Target>
</Project>
```

See also

- Tasks
- Task reference

DownloadFile task

2/21/2019 • 2 minutes to read • Edit Online

Downloads the specified files using the Hyper-Text Transfer Protocol (HTTP).

> **NOTE**
> The DownloadFile task is available in MSBuild 15.8 and above only.

Parameters

The following table describes the parameters of the `DownloadFile` task.

PARAMETER	DESCRIPTION
`DestinationFileName`	Optional ITaskItem parameter The name to use for the downloaded file. By default, the file name is derived from the `SourceUrl` or the remote server.
`DestinationFolder`	Required ITaskItem parameter. Specifies the destination folder to download the file to. If folder is created if it does not exist.
`DownloadedFile`	Optional ITaskItem output parameter. Specifies the file that was downloaded.
`Retries`	Optional `Int32` parameter. Specifies how many times to attempt to download, if all previous attempts have failed. Defaults to zero.
`RetryDelayMilliseconds`	Optional `Int32` parameter. Specifies the delay in milliseconds between any necessary retries. Defaults to 5000.
`SkipUnchangedFiles`	Optional `Boolean` parameter. If `true`, skips the downloading of files that are unchanged. Defaults to `true`. The `DownloadFile` task considers files to be unchanged if they have the same size and the same last modified time according to the remote server. **Note:** Not all HTTP servers indicate the last modified date of files will cause the file to be downloaded again.
`SourceUrl`	Required `String` parameter. Specifies the URL to download.

Remarks

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following example downloads a file and includes it in the `Content` items prior to building the project.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <PropertyGroup>
      <MyUrl>https://raw.githubusercontent.com/Microsoft/msbuild/master/LICENSE</MyUrl>
    </PropertyGroup>

    <Target Name="DownloadContentFiles" BeforeTargets="Build">
        <DownloadFile
            SourceUrl="$(MyUrl)"
            DestinationFolder="$(MSBuildProjectDirectory)">
        <Output TaskParameter="DownloadedFile" ItemName="Content" />
      </DownloadFile>
    </Target>

</Project>
```

See also

- Tasks
- Task reference

Error task

2/21/2019 • 2 minutes to read • Edit Online

Stops a build and logs an error based on an evaluated conditional statement.

Parameters

The following table describes the parameters of the `Error` task.

PARAMETER	DESCRIPTION
`Code`	Optional `String` parameter. The error code to associate with the error.
`File`	Optional `String` parameter. The name of the file that contains the error. If no file name is provided, the file containing the Error task will be used.
`HelpKeyword`	Optional `String` parameter. The Help keyword to associate with the error.
`Text`	Optional `String` parameter. The error text that MSBuild logs if the `Condition` parameter evaluates to `true`.

Remarks

The `Error` task allows MSBuild projects to issue error text to loggers and stop build execution.

If the `Condition` parameter evaluates to `true`, the build is stopped, and an error is logged. If a `Condition` parameter does not exist, the error is logged and build execution stops. For more information on logging, see Obtaining build logs.

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following code example verifies that all required properties are set. If they are not set, the project raises an error event, and logs the value of the `Text` parameter of the `Error` task.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <Target Name="ValidateCommandLine">
        <Error
            Text=" The 0 property must be set on the command line."
            Condition="'$(0)' == ''" />
        <Error
            Text="The FREEBUILD property must be set on the command line."
            Condition="'$(FREEBUILD)' == ''" />
    </Target>
    ...
</Project>
```

See also

- Task reference
- Obtain build logs

Exec task

2/21/2019 • 2 minutes to read • Edit Online

Runs the specified program or command by using the specified arguments.

Parameters

The following table describes the parameters for the `Exec` task.

PARAMETER	DESCRIPTION
`Command`	Required `String` parameter. The command(s) to run. These can be system commands, such as attrib, or an executable, such as *program.exe*, *runprogram.bat*, or *setup.msi*. This parameter can contain multiple lines of commands. Alternatively, you can put multiple commands in a batch file and run it by using this parameter.
`ConsoleOutput`	Optional ITaskItem `[]` output parameter. Each item output is a line from the standard output or standard error stream emitted by the tool. This is only captured if `ConsoleToMsBuild` is set to `true`.
`ConsoleToMsBuild`	Optional `Boolean` parameter. If `true`, the task will capture the standard error and standard output of the tool and make them available in the `ConsoleOutput` output parameter. Default: `false`.
`CustomErrorRegularExpression`	Optional `String` parameter. Specifies a regular expression that is used to spot error lines in the tool output. This is useful for tools that produce unusually formatted output. Default: `null` (no custom processing).
`CustomWarningRegularExpression`	Optional `String` parameter. Specifies a regular expression that is used to spot warning lines in the tool output. This is useful for tools that produce unusually formatted output. Default: `null` (no custom processing).

PARAMETER	DESCRIPTION
`EchoOff`	Optional `Boolean` parameter. If `true`, the task will not emit the expanded form of `Command` to the MSBuild log. Default: `false`.
`ExitCode`	Optional `Int32` output read-only parameter. Specifies the exit code that is provided by the executed command.
`IgnoreExitCode`	Optional `Boolean` parameter. If `true`, the task ignores the exit code that is provided by the executed command. Otherwise, the task returns `false` if the executed command returns a non-zero exit code. Default: `false`.
`IgnoreStandardErrorWarningFormat`	Optional `Boolean` parameter. If `false`, selects lines in the output that match the standard error/warning format, and logs them as errors/warnings. If `true`, disable this behavior. Default: `false`.
`Outputs`	Optional ITaskItem `[]` output parameter. Contains the output items from the task. The `Exec` task does not set these itself. Instead, you can provide them as if it did set them, so that they can be used later in the project.
`StdErrEncoding`	Optional `String` output parameter. Specifies the encoding of the captured task standard error stream. The default is the current console output encoding.
`StdOutEncoding`	Optional `String` output parameter. Specifies the encoding of the captured task standard output stream. The default is the current console output encoding.
`WorkingDirectory`	Optional `String` parameter. Specifies the directory in which the command will run. Default: The project's current working directory.

Remarks

This task is useful when a specific MSBuild task for the job that you want to perform is not available. However, the `Exec` task, unlike a more specific task, cannot do additional processing or conditional operations based on the result of the tool or command that it runs.

The `Exec` task calls *cmd.exe* instead of directly invoking a process.

In addition to the parameters listed in this document, this task inherits parameters from the ToolTaskExtension class, which itself inherits from the ToolTask class. For a list of these additional parameters and their descriptions, see ToolTaskExtension base class.

Example

The following example uses the `Exec` task to run a command.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <ItemGroup>
        <Binaries Include="*.dll;*.exe"/>
    </ItemGroup>

    <Target Name="SetACL">
        <!-- set security on binaries-->
        <Exec Command="echo y| cacls %(Binaries.Identity) /G everyone:R"/>
    </Target>
</Project>
```

See also

- Tasks
- Task reference

FindAppConfigFile task

2/21/2019 • 2 minutes to read • Edit Online

Finds the *app.config* file, if any, in the provided lists.

Parameters

The following table describes the parameters of the `FindAppConfigFile` task.

PARAMETER	DESCRIPTION
`AppConfigFile`	Optional ITaskItem `[]` output parameter. Specifies the first matching item found in the list, if any.
`PrimaryList`	Required ITaskItem `[]` parameter. Specifies the primary list to search through.
`SecondaryList`	Required ITaskItem `[]` parameter. Specifies the secondary list to search through.
`TargetPath`	Required `String` parameter. Specifies the value to add as metadata.

Remarks

In addition to having the parameters that are listed in the table, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

See also

- Tasks
- Task reference

FindInList task

2/21/2019 • 2 minutes to read • Edit Online

In a specified list, finds an item that has the matching itemspec.

Parameters

The following table describes the parameters of the FindInList task.

PARAMETER	DESCRIPTION
`CaseSensitive`	Optional `Boolean` parameter. If `true`, the search is case-sensitive; otherwise, it is not. Default value is `true`.
`FindLastMatch`	Optional `Boolean` parameter. If `true`, return the last match; otherwise, return the first match. Default value is `false`.
`ItemFound`	Optional ITaskItem `[]` read-only output parameter. The first matching item found in the list, if any.
`ItemSpecToFind`	Required `String` parameter. The itemspec to search for.
`List`	Required ITaskItem `[]` parameter. The list in which to search for the itemspec.
`MatchFileNameOnly`	Optional `Boolean` parameter. If `true`, match against just the file name part of the itemspec; otherwise, match against the whole itemspec. Default value is `true`.

Remarks

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

See also

- Tasks
- Task reference

FindUnderPath task

2/21/2019 • 2 minutes to read • Edit Online

Determines which items in the specified item collection have paths that are in or below the specified folder.

Parameters

The following table describes the parameters of the `FindUnderPath` task.

PARAMETER	DESCRIPTION
`Files`	Optional ITaskItem `[]` parameter. Specifies the files whose paths should be compared with the path specified by the `Path` parameter.
`InPath`	Optional ITaskItem `[]` output parameter. Contains the items that were found under the specified path.
`OutOfPath`	Optional ITaskItem `[]` output parameter. Contains the items that were not found under the specified path.
`Path`	Required ITaskItem parameter. Specifies the folder path to use as the reference.
`UpdateToAbsolutePaths`	Optional `Boolean` parameter. If true, the paths of the output items are updated to be absolute paths.

Remarks

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following example uses the `FindUnderPath` task to determine if the files contained in the `MyFiles` item have paths that exist under the path specified by the `SearchPath` property. After the task completes, the `FilesNotFoundInPath` item contains the *File1.txt* file, and the `FilesFoundInPath` item contains the *File2.txt* file.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <ItemGroup>
        <MyFiles Include="C:\File1.txt" />
        <MyFiles Include="C:\Projects\MyProject\File2.txt" />
    </ItemGroup>

    <PropertyGroup>
        <SearchPath>C:\Projects\MyProject</SearchPath>
    </PropertyGroup>

    <Target Name="FindFiles">
        <FindUnderPath
            Files="@(MyFiles)"
            Path="$(SearchPath)">
            <Output
                TaskParameter="InPath"
                ItemName="FilesFoundInPath" />
            <Output
                TaskParameter="OutOfPath"
                ItemName="FilesNotFoundInPath" />
        </FindUnderPath>
    </Target>

</Project>
```

See also

- Task reference
- Tasks
- MSBuild concepts

FormatUrl task

2/21/2019 • 2 minutes to read • Edit Online

Converts a URL to a correct URL format.

Parameters

The following table describes the parameters of the `FormatUrl` task.

PARAMETER	DESCRIPTION
`InputUrl`	Optional `String` parameter. Specifies the URL to format.
`OutputUrl`	Optional `String` output parameter. Specifies the formatted URL.

Remarks

In addition to having the parameters that are listed in the table, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

See also

- Tasks
- Task reference

FormatVersion task

4/18/2019 • 2 minutes to read • Edit Online

Appends the revision number to the version number.

- Case #1: Input: Version=<undefined>; Revision=<don't care>; Output: OutputVersion="1.0.0.0"
- Case #2: Input: Version="1.0.0.*" Revision="5" Output: OutputVersion="1.0.0.5"
- Case #3: Input: Version="1.0.0.0" Revision=<don't care>; Output: OutputVersion="1.0.0.0"

Parameters

The following table describes the parameters of the `FormatVersion` task.

PARAMETER	DESCRIPTION
`FormatType`	Optional `String` parameter. Specifies the format type. - "Version" = version. - "Path" = replace "." with "_";
`OutputVersion`	Optional `String` output parameter. Specifies the output version that includes the revision number.
`Revision`	Optional `Int32` parameter. Specifies the revision to append to the version.
`Version`	Optional `String` parameter. Specifies the version number string to format.

Remarks

In addition to having the parameters that are listed in the table, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

See also

- Tasks
- Task reference

GenerateApplicationManifest task

2/21/2019 • 10 minutes to read • Edit Online

Generates a ClickOnce application manifest or a native manifest. A native manifest describes a component by defining a unique identity for the component and identifying all assemblies and files that make up the component. A ClickOnce application manifest extends a native manifest by indicating the entry point of the application, and specifying the application security level.

Parameters

The following table describes the parameters for the `GenerateApplicationManifest` task.

PARAMETER	DESCRIPTION
`AssemblyName`	Optional `String` parameter. Specifies the `Name` field of the assembly identity for the generated manifest. If this parameter is not specified, the name is inferred from the `EntryPoint` or `InputManifest` parameters. If no name can be created, the task throws an error.
`AssemblyVersion`	Optional `String` parameter. Specifies the `Version` field of the assembly identity for the generated manifest. If this parameter is not specified, a default value of "1.0.0.0" is used.
`ClrVersion`	Optional `String` parameter. Specifies the minimum version of the Common Language Runtime (CLR) required by the application. The default value is the CLR version in use by the build system. If the task is generating a native manifest, this parameter is ignored.
`ConfigFile`	Optional ITaskItem `[]` parameter. Specifies which item contains the application configuration file. If the task is generating a native manifest, this parameter is ignored.
`Dependencies`	Optional ITaskItem `[]` parameter. Specifies an item list that defines the set of dependent assemblies for the generated manifest. Each item may be further described by item metadata to indicate additional deployment state and the type of dependence. For more information, see Item metadata.
`Description`	Optional `String` parameter. Specifies the description for the application or component.

PARAMETER	DESCRIPTION
`EntryPoint`	Optional ITaskItem `[]` parameter. Specifies a single item that indicates the entry point for the generated manifest assembly. For a ClickOnce application manifest, this parameter specifies the assembly that starts when the application is run.
`ErrorReportUrl`	Optional System.String parameter. Specifies the URL of the web page that is displayed in dialog boxes during error reports in ClickOnce installations.
`FileAssociations`	Optional ITaskItem `[]` parameter. Specifies a list of one or more file type that are associated with the ClickOnce deployment manifest. File associations only valid only when .NET Framework 3.5 or later is targeted.
`Files`	Optional ITaskItem `[]` parameter. The files to include in the manifest. Specify the full path for each file.
`HostInBrowser`	Optional Boolean parameter. If `true`, the application is hosted in a browser (as are WPF Web Browser Applications).
`IconFile`	Optional ITaskItem `[]` parameter. Indicates the application icon file. The application icon is expressed in the generated application manifest and is used for the **Start Menu** and **Add/Remove Programs** dialog. If this input is not specified, a default icon is used. If the task is generating a native manifest, this parameter is ignored.
`InputManifest`	Optional ITaskItem parameter. Indicates an input XML document to serve as a base for the manifest generator. This allows structured data such as application security or custom manifest definitions to be reflected in the output manifest. The root element in the XML document must be an assembly node in the asmv1 namespace.
`IsolatedComReferences`	Optional ITaskItem `[]` parameter. Specifies COM components to isolate in the generated manifest. This parameter supports the ability to isolate COM components for "Registration Free COM" deployment. It works by auto-generating a manifest with standard COM registration definitions. However, the COM components must be registered on the build machine in order for this to function properly.

PARAMETER	DESCRIPTION
`ManifestType`	Optional `String` parameter. Specifies which type of manifest to generate. This parameter can have the following values: - `Native` - `ClickOnce` If this parameter is not specified, the task defaults to `ClickOnce`.
`MaxTargetPath`	Optional `String` parameter. Specifies the maximum allowable length of a file path in a ClickOnce application deployment. If this value is specified, the length of each file path in the application is checked against this limit. Any items that exceed the limit will raise in a build warning. If this input is not specified or is zero, then no checking is performed. If the task is generating a native manifest, this parameter is ignored.
`OSVersion`	Optional `String` parameter. Specifies the minimum required operating system (OS) version required by the application. For example, the value "5.1.2600.0" indicates the operating system is Windows XP. If this parameter is not specified, the value "4.10.0.0" is used, which indicates Windows 98 Second Edition, the minimum supported OS of the .NET Framework. If the task is generating a native manifest, this input is ignored.
`OutputManifest`	Optional ITaskItem output parameter. Specifies the name of the generated output manifest file. If this parameter is not specified, the name of the output file is inferred from the identity of the generated manifest.
`Platform`	Optional `String` parameter. Specifies the target platform of the application. This parameter can have the following values: - `AnyCPU` - `x86` - `x64` - `Itanium` If this parameter is not specified, the task defaults to `AnyCPU`.
`Product`	Optional `String` parameter. Specifies the name of the application. If this parameter is not specified, the name is inferred from the identity of the generated manifest. This name is used for the shortcut name on the **Start** menu and is part of the name that appears in the **Add or Remove Programs** dialog box.

PARAMETER	DESCRIPTION
`Publisher`	Optional `String` parameter. Specifies the publisher of the application. If this parameter is not specified, the name is inferred from the registered user, or the identity of the generated manifest. This name is used for the folder name on the **Start** menu and is part of the name that appears in the **Add or Remove Programs** dialog box.
`RequiresMinimumFramework35SP1`	Optional `Boolean` parameter. If true, the application requires the .NET Framework 3.5 SP1 or a more recent version.
`TargetCulture`	Optional `String` parameter. Identifies the culture of the application and specifies the `Language` field of the assembly identity for the generated manifest. If this parameter is not specified, it is assumed the application is culture invariant.
`TargetFrameworkMoniker`	Optional `String` parameter. Specifies the target framework moniker.
`TargetFrameworkProfile`	Optional `String` parameter. Specifies the target framework profile.
`TargetFrameworkSubset`	Optional `String` parameter. Specifies the name of the .NET Framework subset to target.
`TargetFrameworkVersion`	Optional `String` parameter. Specifies the target .NET Framework of the project.
`TrustInfoFile`	Optional ITaskItem parameter. Indicates an XML document that specifies the application security. The root element in the XML document must be a trustInfo node in the asmv2 namespace. If the task is generating a native manifest, this parameter is ignored.
`UseApplicationTrust`	Optional `Boolean` parameter. If true, the `Product`, `Publisher`, and `SupportUrl` properties are written to the application manifest.

Remarks

In addition to the parameters listed above, this task inherits parameters from the GenerateManifestBase class, which itself inherits from the Task class. For a list of the parameters of the Task class, see Task base class.

For information about how to use the `GenerateDeploymentManifest` task, see GenerateApplicationManifest task.

The inputs for dependencies and files may be further decorated with item metadata to specify additional

deployment state for each item.

Item metadata

METADATA NAME	DESCRIPTION
`DependencyType`	Indicates whether the dependency is published and installed with the application or a prerequisite. This metadata is valid for all dependencies, but is not used for files. The available values for this metadata are: - `Install` - `Prerequisite` Install is the default value.
`AssemblyType`	Indicates whether the dependency is a managed or a native assembly. This metadata is valid for all dependencies, but is not used for files. The available values for this metadata are: - `Managed` - `Native` - `Unspecified` `Unspecified` is the default value, which indicates that the manifest generator will determine the assembly type automatically.
`Group`	Indicates the group for downloading additional files on-demand. The group name is defined by the application and can be any string. An empty string indicates the file is not part of a download group, which is the default. Files not in a group are part of the initial application download. Files in a group are only downloaded when explicitly requested by the application using System.Deployment.Application. This metadata is valid for all files where `IsDataFile` is `false` and all dependencies where `DependencyType` is `Install`.
`TargetPath`	Specifies how the path should be defined in the generated manifest. This attribute is valid for all files. If this attribute is not specified, the item specification is used. This attribute is valid for all files and dependencies with a `DependencyType` value of `Install`.
`IsDataFile`	A `Boolean` metadata value that indicates whether or not the file is a data file. A data file is special in that it is migrated between application updates. This metadata is only valid for files. `False` is the default value.

Example

This example uses the `GenerateApplicationManifest` task to generate a ClickOnce application manifest and the `GenerateDeploymentManifest` task to generate a deployment manifest for an application with a single assembly. It then uses the `SignFile` task to sign the manifests.

This illustrates the simplest possible manifest generation scenario where ClickOnce manifests are generated for a

single program. A default name and identity are inferred from the assembly for the manifest.

> **NOTE**
> In the example below, all application binaries are pre-built in order to focus on manifest generation aspects. This example produces a fully working ClickOnce deployment.

> **NOTE**
> For more information on the `Thumbprint` property used in the `SignFile` task in this example, see SignFile task.

```
<Project DefaultTargets="Build"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <ItemGroup>
        <EntryPoint Include="SimpleWinApp.exe" />
    </ItemGroup>

    <PropertyGroup>
        <Thumbprint>
             <!-- Insert generated thumbprint here -->
        </Thumbprint>
    </PropertyGroup>

    <Target Name="Build">

        <GenerateApplicationManifest
            EntryPoint="@(EntryPoint)">
            <Output
                ItemName="ApplicationManifest"
                TaskParameter="OutputManifest"/>
        </GenerateApplicationManifest>

        <GenerateDeploymentManifest
            EntryPoint="@(ApplicationManifest)">
            <Output
                ItemName="DeployManifest"
                TaskParameter="OutputManifest"/>
        </GenerateDeploymentManifest>

        <SignFile
            CertificateThumbprint="$(Thumbprint)"
            SigningTarget="@(ApplicationManifest)"/>

        <SignFile
            CertificateThumbprint="$(Thumbprint)"
            SigningTarget="@(DeployManifest)"/>

    </Target>
</Project>
```

Example

This example uses the `GenerateApplicationManifest` and `GenerateDeploymentManifest` tasks to generate ClickOnce application and deployment manifests for an application with a single assembly, specifying name and identity of manifests.

This example is similar to previous example except the name and identity of the manifests are explicitly specified. Also, this example is configured as an online application instead of an installed application.

NOTE

In the example below, all application binaries are pre-built in order to focus on manifest generation aspects. This example produces a fully working ClickOnce deployment.

NOTE

For more information on the `Thumbprint` property used in the `SignFile` task in this example, see SignFile task.

```
<Project DefaultTargets="Build"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <ItemGroup>
        <EntryPoint Include="SimpleWinApp.exe" />
    </ItemGroup>

    <PropertyGroup>
        <Thumbprint>
             <!-- Insert generated thumbprint here -->
        </Thumbprint>
    </PropertyGroup>

    <Target Name="Build">

        <GenerateApplicationManifest
            AssemblyName="SimpleWinApp.exe"
            AssemblyVersion="1.0.0.0"
            EntryPoint="@(EntryPoint)"
            OutputManifest="SimpleWinApp.exe.manifest">
            <Output
                ItemName="ApplicationManifest"
                TaskParameter="OutputManifest"/>
        </GenerateApplicationManifest>

        <GenerateDeploymentManifest
                AssemblyName="SimpleWinApp.application"
                AssemblyVersion="1.0.0.0"
                EntryPoint="@(ApplicationManifest)"
                Install="false"
                OutputManifest="SimpleWinApp.application">
                <Output
                    ItemName="DeployManifest"
                    TaskParameter="OutputManifest"/>
        </GenerateDeploymentManifest>

        <SignFile
            CertificateThumbprint="$(Thumbprint)"
            SigningTarget="@(ApplicationManifest)"/>

        <SignFile
            CertificateThumbprint="$(Thumbprint)"
            SigningTarget="@(DeployManifest)"/>

    </Target>
</Project>
```

Example

This example uses the `GenerateApplicationManifest` and `GenerateDeploymentManifest` tasks to generate ClickOnce application and deployment manifests for an application with multiple files and assemblies.

NOTE

In the example below, all application binaries are pre-built in order to focus on manifest generation aspects. This example produces a fully working ClickOnce deployment.

NOTE

For more information on the `Thumbprint` property used in the `SignFile` task in this example, see SignFile task.

```
<Project DefaultTargets="Build"
    xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <ItemGroup>
        <EntryPoint Include="SimpleWinApp.exe" />
    </ItemGroup>

    <PropertyGroup>
        <Thumbprint>
             <!-- Insert generated thumbprint here -->
        </Thumbprint>
        <DeployUrl>
            <!-- Insert the deployment URL here -->
        </DeployUrl>
        <SupportUrl>
            <!-- Insert the support URL here -->
        </SupportUrl>
    </PropertyGroup>

    <Target Name="Build">

    <ItemGroup>
        <EntryPoint Include="SimpleWinApp.exe"/>
        <Dependency Include="ClassLibrary1.dll">
            <AssemblyType>Managed</AssemblyType>
            <DependencyType>Install</DependencyType>
        </Dependency>
        <Dependency Include="ClassLibrary2.dll">
            <AssemblyType>Managed</AssemblyType>
            <DependencyType>Install</DependencyType>
            <Group>Secondary</Group>
        </Dependency>
        <Dependency Include="MyAddIn1.dll">
            <AssemblyType>Managed</AssemblyType>
            <DependencyType>Install</DependencyType>
            <TargetPath>Addins\MyAddIn1.dll</TargetPath>
        </Dependency>
        <Dependency Include="ClassLibrary3.dll">
            <AssemblyType>Managed</AssemblyType>
            <DependencyType>Prerequisite</DependencyType>
        </Dependency>

        <File Include="Text1.txt">
            <TargetPath>Text\Text1.txt</TargetPath>
            <Group>Text</Group>
        </File>
        <File Include="DataFile1.xml ">
            <TargetPath>Data\DataFile1.xml</TargetPath>
            <IsDataFile>true</IsDataFile>
        </File>

        <IconFile Include="Heart.ico"/>
        <ConfigFile Include="app.config">
            <TargetPath>SimpleWinApp.exe.config</TargetPath>
        </ConfigFile>
```

```
        </ConfigFile>
        <BaseManifest Include="app.manifest"/>
    </ItemGroup>

    <Target Name="Build">

        <GenerateApplicationManifest
            AssemblyName="SimpleWinApp.exe"
            AssemblyVersion="1.0.0.0"
            ConfigFile="@(ConfigFile)"
            Dependencies="@(Dependency)"
            Description="TestApp"
            EntryPoint="@(EntryPoint)"
            Files="@(File)"
            IconFile="@(IconFile)"
            InputManifest="@(BaseManifest)"
            OutputManifest="SimpleWinApp.exe.manifest">
            <Output
                ItemName="ApplicationManifest"
                TaskParameter="OutputManifest"/>
        </GenerateApplicationManifest>

        <GenerateDeploymentManifest
            AssemblyName="SimpleWinApp.application"
            AssemblyVersion="1.0.0.0"
            DeploymentUrl="$(DeployToUrl)"
            Description="TestDeploy"
            EntryPoint="@(ApplicationManifest)"
            Install="true"
            OutputManifest="SimpleWinApp.application"
            Product="SimpleWinApp"
            Publisher="Microsoft"
            SupportUrl="$(SupportUrl)"
            UpdateEnabled="true"
            UpdateInterval="3"
            UpdateMode="Background"
            UpdateUnit="weeks">
            <Output
                ItemName="DeployManifest"
                TaskParameter="OutputManifest"/>
        </GenerateDeploymentManifest>

        <SignFile
            CertificateThumbprint="$(Thumbprint)"
            SigningTarget="@(ApplicationManifest)"/>

        <SignFile
            CertificateThumbprint="$(Thumbprint)"
            SigningTarget="@(DeployManifest)"/>

    </Target>
</Project>
```

Example

This example uses the `GenerateApplicationManifest` task to generate a native manifest for application *Test.exe*, referencing native component *Alpha.dll* and an isolated COM component *Bravo.dll*.

This example produces the *Test.exe.manifest*, making the application XCOPY deployable and taking advantage of Registration Free COM.

> **NOTE**
>
> In the example below, all application binaries are pre-built in order to focus on manifest generation aspects. This example produces a fully working ClickOnce deployment.

```
<Project DefaultTargets="Build" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <ItemGroup>
        <File Include="Test.exe" />
        <Dependency Include="Alpha.dll">
            <AssemblyType>Native</AssemblyType>
            <DependencyType>Install</DependencyType>
        </Dependency>
        <ComComponent Include="Bravo.dll" />
    </ItemGroup>

    <Target Name="Build">
        <GenerateApplicationManifest
            AssemblyName="Test.exe"
            AssemblyVersion="1.0.0.0"
            Dependencies="@(Dependency)"
            Files="@(File)"
            IsolatedComReferences="@(ComComponent)"
            ManifestType="Native">
            <Output
                ItemName="ApplicationManifest"
                TaskParameter="OutputManifest"/>
        </GenerateApplicationManifest>

    </Target>
</Project>
```

See also

- Tasks
- GenerateDeploymentManifest task
- SignFile task
- Task reference

GenerateBootstrapper task

6/6/2019 • 3 minutes to read • Edit Online

Provides an automated way to detect, download, and install an application and its prerequisites. It serves as a single installer that integrates the separate installers for all the components making up an application.

Task parameters

The following describe the parameters of the `GenerateBootstrapper` task.

- `ApplicationFile`

  Optional `String` parameter.

  Specifies the file the bootstrapper will use to begin the installation of the application after all prerequisites have been installed. A build error will result if neither the `BootstrapperItems` nor the `ApplicationFile` parameter is specified.

- `ApplicationName`

  Optional `String` parameter.

  Specifies the name of the application that the bootstrapper will install. This name will appear in the UI the bootstrapper uses during installation.

- `ApplicationRequiresElevation`

  Optional `Boolean` parameter.

  If `true`, the component runs with elevated permissions when it is installed on a target computer.

- `ApplicationUrl`

  Optional `String` parameter.

  Specifies the Web location that is hosting the application's installer.

- `BootstrapperComponentFiles`

  Optional `String[]` output parameter.

  Specifies the built location of bootstrapper package files.

- `BootstrapperItems`

  Optional ITaskItem `[]` parameter.

  Specifies the products to build into the bootstrapper. The items passed to this parameter should have the following syntax:

```
<BootstrapperItem
    Include="ProductCode">
    <ProductName>
        ProductName
    </ProductName>
</BootstrapperItem>
```

The `Include` attribute represents the name of a prerequisite that should be installed. The `ProductName` item metadata is optional, and will be used by the build engine as a user-friendly name if the package cannot be found. These items are not required MSBuild input parameters, unless no `ApplicationFile` is specified. You should include one item for every prerequisite that must be installed for your application.

A build error will result if neither the `BootstrapperItems` nor the `ApplicationFile` parameter is specified.

- `BootstrapperKeyFile`

  Optional `String` output parameter.

  Specifies the built location of *setup.exe*

- `ComponentsLocation`

  Optional `String` parameter.

  Specifies a location for the bootstrapper to look for installation prerequisites to install. This parameter can have the following values:

  - `HomeSite`: Indicates that the prerequisite is being hosted by the component vendor.
  - `Relative`: Indicates that the prerequisite is at the same location of the application.
  - `Absolute`: Indicates that all components are to be found at a centralized URL. This value should be used in conjunction with the `ComponentsUrl` input parameter.

    If `ComponentsLocation` is not specified, `HomeSite` is used by default.

- `ComponentsUrl`

  Optional `String` parameter.

  Specifies the URL containing the installation prerequisites.

- `CopyComponents`

  Optional `Boolean` parameter.

  If `true`, the bootstrapper copies all output files to the path specified in the `OutputPath` parameter. The values of the `BootstrapperComponentFiles` parameter should all be based on this path. If `false`, the files are not copied, and the `BootstrapperComponentFiles` values are based on the value of the `Path` parameter. The default value of this parameter is `true`.

- `Culture`

  Optional `String` parameter.

  Specifies the culture to use for the bootstrapper UI and installation prerequisites. If the specified culture is unavailable, the task uses the value of the `FallbackCulture` parameter.

- `FallbackCulture`

  Optional `String` parameter.

  Specifies the secondary culture to use for the bootstrapper UI and installation prerequisites.

- `OutputPath`

  Optional `String` parameter.

  Specifies the location to copy *setup.exe* and all package files.

- `Path`

  Optional `String` parameter.

  Specifies the location of all available prerequisite packages.

- `SupportUrl`

  Optional `String` parameter.

  Specifies the URL to provide if the bootstrapper installation fails.

- `Validate`

  Optional `Boolean` parameter.

  If `true`, the bootstrapper performs XSD validation on the specified input bootstrapper items. The default value of this parameter is `false`.

Remarks

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following example uses the `GenerateBootstrapper` task to install an application that must have the .NET Framework 2.0 installed as a prerequisite.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <ItemGroup>
        <BootstrapperFile Include="Microsoft.Net.Framework.2.0">
            <ProductName>Microsoft .NET Framework 2.0</ProductName>
        </BootstrapperFile>
    </ItemGroup>

    <Target Name="BuildBootstrapper">
        <GenerateBootstrapper
            ApplicationFile="WindowsApplication1.application"
            ApplicationName="WindowsApplication1"
            ApplicationUrl="http://mycomputer"
            BootstrapperItems="@(BootstrapperFile)"
            OutputPath="C:\output" />
    </Target>

</Project>
```

See also

- Tasks
- Task reference

GenerateDeploymentManifest task

4/16/2019 • 5 minutes to read • Edit Online

Generates a ClickOnce deployment manifest. A ClickOnce deployment manifest describes the deployment of an application by defining a unique identity for the deployment, identifying deployment traits such as install or online mode, specifying application update settings and update locations, and indicating the corresponding ClickOnce application manifest.

Parameters

The following table describes the parameters for the `GenerateDeploymentManifest` task.

PARAMETER	DESCRIPTION
`AssemblyName`	Optional `String` parameter. Specifies the `Name` field of the assembly identity for the generated manifest. If this parameter is not specified, the name is inferred from the `EntryPoint` or `InputManifest` parameters. If the name cannot be inferred, the task throws an error.
`AssemblyVersion`	Optional `String` parameter. Specifies the `Version` field of the assembly identity for the generated manifest. If this parameter is not specified, the task uses the value "1.0.0.0".
`CreateDesktopShortcut`	Optional `Boolean` parameter. If true, an icon is created on the desktop during ClickOnce application installation.
`DeploymentUrl`	Optional `String` parameter. Specifies the update location for the application. If this parameter is not specified, no update location is defined for the application. However, if the `UpdateEnabled` parameter is `true`, the update location must be specified. The specified value should be a fully qualified URL or UNC path.
`Description`	Optional `String` parameter. Specifies an optional description for the application.
`DisallowUrlActivation`	Optional `Boolean` parameter. Specifies whether the application should be run automatically when it is opened through a URL. If this parameter is `true`, the application can only be started from the **Start** menu. The default value of this parameter is `false`. This input applies only when the `Install` parameter value is `true`.

PARAMETER	DESCRIPTION

`EntryPoint`	Optional ITaskItem `[]` parameter.
	
	Indicates the entry point for the generated manifest assembly. For a ClickOnce deployment manifest, this input specifies the ClickOnce application manifest.
	
	If the `EntryPoint` task parameter is not specified, the `<customHostSpecified>` tag is inserted as a child of the `<entryPoint>` tag, for example:
	
	`<entryPoint xmlns="urn:schemas-microsoft-com:asm.v2">`
	
	`<co.v1:customHostSpecified />`
	
	`</entryPoint>`
	
	You can add DLL dependencies to the application manifest by using the following steps:
	
	1. Resolve the assembly references with a call to ResolveAssemblyReference.
	2. Pass the output of the previous task and the assembly itself to ResolveManifestFiles.
	3. Pass the dependencies by using the `Dependencies` parameter to GenerateApplicationManifest.

`ErrorReportUrl`	Optional System.String parameter. Specifies the URL of the web page that is displayed in dialog boxes during ClickOnce installations.
`InputManifest`	Optional ITaskItem parameter. Indicates an input XML document to serve as a base for the manifest generator. This enables structured data, such as custom manifest definitions, to be reflected in the output manifest. The root element in the XML document must be an assembly node in the asmv1 namespace.
`Install`	Optional `Boolean` parameter. Specifies whether the application is an installed application or an online-only application. If this parameter is `true`, the application will be installed on the user's **Start** menu, and can be removed by using the **Add or Remove Programs** dialog box. If this parameter is `false`, the application is intended for online use from a web page. The default value of this parameter is `true`.

PARAMETER	DESCRIPTION
`MapFileExtensions`	Optional `Boolean` parameter. Specifies whether the *.deploy* file name extension mapping is used. If this parameter is `true`, every program file is published with a *.deploy* file name extension. This option is useful for web server security to limit the number of file name extensions that must be unblocked to enable ClickOnce application deployment. The default value of this parameter is `false`.
`MaxTargetPath`	Optional `String` parameter. Specifies the maximum allowed length of a file path in a ClickOnce application deployment. If this parameter is specified, the length of each file path in the application is checked against this limit. Any items that exceed the limit will cause a build warning. If this input is not specified or is zero, no checking is performed.
`MinimumRequiredVersion`	Optional `String` parameter. Specifies whether the user can skip the update. If the user has a version that is less than the minimum required, he will not have the option to skip the update. This input only applies when the value of the `Install` parameter is `true`.
`OutputManifest`	Optional ITaskItem parameter. Specifies the name of the generated output manifest file. If this parameter is not specified, the name of the output file is inferred from the identity of the generated manifest.
`Platform`	Optional `String` parameter. Specifies the target platform of the application. This parameter can have the following values: - `AnyCPU` - `x86` - `x64` - `Itanium` The default value is `AnyCPU`.
`Product`	Optional `String` parameter. Specifies the name of the application. If this parameter is not specified, the name is inferred from the identity of the generated manifest. This name is used for the shortcut name on the **Start** menu and is part of the name that appears in the **Add or Remove Programs** dialog box.

PARAMETER	DESCRIPTION
`Publisher`	Optional `String` parameter. Specifies the publisher of the application. If this parameter is not specified, the name is inferred from the registered user, or the identity of the generated manifest. This name is used for the folder name on the **Start** menu and is part of the name that appears in the **Add or Remove Programs** dialog box.
`SuiteNamel`	Optional `String` parameter. Specifies the name of the folder on the **Start** menu where the application is located after ClickOnce deployment.
`SupportUrl`	Optional `String` parameter. Specifies the link that appears in the **Add or Remove Programs** dialog box for the application. The specified value should be a fully qualified URL or UNC path.
`TargetCulture`	Optional `String` parameter. Identifies the culture of the application, and specifies the `Language` field of the assembly identity for the generated manifest. If this parameter is not specified, it is assumed that the application is culture invariant.
`TrustUrlParameters`	Optional `Boolean` parameter. Specifies whether URL query-string parameters should be made available to the application. The default value of this parameter is `false`, which indicates that parameters will not be available to the application.
`UpdateEnabled`	Optional `Boolean` parameter. Indicates whether the application is enabled for updates. The default value of this parameter is `false`. This parameter only applies when the value of the `Install` parameter is `true`.
`UpdateInterval`	Optional `Int32` parameter. Specifies the update interval for the application. The default value of this parameter is zero. This parameter only applies when the values of the `Install` and `UpdateEnabled` parameters are both `true`.

PARAMETER	DESCRIPTION
`UpdateMode`	Optional `String` parameter. Specifies whether updates should be checked in the foreground before the application is started, or in the background as the application is running. This parameter can have the following values: - `Foreground` - `Background` The default value of this parameter is `Background`. This parameter only applies when the values of the `Install` and `UpdateEnabled` parameters are both `true`.
`UpdateUnit`	Optional `String` parameter. Specifies the units for the `UpdateInterval` parameter. This parameter can have the following values: - `Hours` - `Days` - `Weeks` This parameter only applies when the values of the `Install` and `UpdateEnabled` parameters are both `true`.

Remarks

In addition to the parameters listed above, this task inherits parameters from the GenerateManifestBase class, which itself inherits from the Task class. For a list of the parameters of the Task class, see Task base class.

See also

- Tasks
- GenerateApplicationManifest task
- SignFile task
- Task reference

GenerateResource task

2/21/2019 • 6 minutes to read • Edit Online

Converts between *.txt* and *.resx* (XML-based resource format) files and common language runtime binary *.resources* files that can be embedded in a runtime binary executable or compiled into satellite assemblies. This task is typically used to convert *.txt* or *.resx* files to *.resources* files. The `GenerateResource` task is functionally similar to resgen.exe.

Parameters

The following table describes the parameters of the `GenerateResource` task.

PARAMETER	DESCRIPTION
`AdditionalInputs`	Optional ITaskItem `[]` parameter. Contains additional inputs to the dependency checking done by this task. For example, the project and targets files typically should be inputs, so that if they are updated, all resources are regenerated.
`EnvironmentVariables`	Optional `String[]` parameter. Specifies an array of name-value pairs of environment variables that should be passed to the spawned *resgen.exe*, in addition to (or selectively overriding) the regular environment block.
`ExcludedInputPaths`	Optional ITaskItem `[]` parameter. Specifies an array of items that specify paths from which tracked inputs will be ignored during Up to date checking.
`ExecuteAsTool`	Optional `Boolean` parameter. If `true`, runs *tlbimp.exe* and *aximp.exe* from the appropriate target framework out-of-proc to generate the necessary wrapper assemblies. This parameter allows multi-targeting of `ResolveComReferences`.
`FilesWritten`	Optional ITaskItem `[]` output parameter. Contains the names of all files written to disk. This includes the cache file, if any. This parameter is useful for implementations of Clean.
`MinimalRebuildFromTracking`	Optional `Boolean` parameter. Gets or sets a switch that specifies whether tracked incremental build will be used. If `true`, incremental build is turned on; otherwise, a rebuild will be forced.

PARAMETER	DESCRIPTION
`NeverLockTypeAssemblies`	Optional `Boolean` parameter. Gets or sets a Boolean value that specifies whether to create a new AppDomain to evaluate the resources (*.resx*) files (true) or to create a new AppDomain only when the resources files reference a user's assembly (false).
`OutputResources`	Optional ITaskItem `[]` output parameter. Specifies the name of the generated files, such as *.resources* files. If you do not specify a name, the name of the matching input file is used and the *.resources* file that is created is placed in the directory that contains the input file.
`PublicClass`	Optional `Boolean` parameter. If `true`, creates a strongly typed resource class as a public class.
`References`	Optional `String[]` parameter. References to load types in *.resx* files from. *.resx* file data elements may have a .NET type. When the *.resx* file is read, this must be resolved. Typically, it is resolved successfully by using standard type loading rules. If you provide assemblies in `References`, they take precedence. This parameter is not required for strongly typed resources.
`SdkToolsPath`	Optional `String` parameter. Specifies the path to the SDK tools, such as *resgen.exe*.
`Sources`	Required ITaskItem `[]` parameter. Specifies the items to convert. Items passed to this parameter must have one of the following file extensions: - *.txt*: Specifies the extension for a text file to convert. Text files can only contain string resources. - *.resx*: Specifies the extension for an XML-based resource file to convert. - *.restext*: Specifies the same format as *.txt*. This different extension is useful if you want to clearly distinguish source files that contain resources from other source files in your build process. - *.resources*: Specifies the extension for a resource file to convert.
`StateFile`	Optional ITaskItem parameter. Specifies the path to an optional cache file that is used to speed up dependency checking of links in *.resx* input files.

PARAMETER	DESCRIPTION
`StronglyTypedClassName`	Optional `String` parameter. Specifies the class name for the strongly typed resource class. If this parameter is not specified, the base name of the resource file is used.
`StronglyTypedFilename`	Optional ITaskItem parameter. Specifies the filename for the source file. If this parameter is not specified, the name of the class is used as the base filename, with the extension dependent on the language. For example: *MyClass.cs*.
`StronglyTypedLanguage`	Optional `String` parameter. Specifies the language to use when generating the class source for the strongly typed resource. This parameter must match exactly one of the languages used by the CodeDomProvider. For example: `VB` or `C#`. By passing a value to this parameter, you instruct the task to generate strongly typed resources.
`StronglyTypedManifestPrefix`	Optional `String` parameter. Specifies the resource namespace or manifest prefix to use in the generated class source for the strongly typed resource.
`StronglyTypedNamespace`	Optional `String` parameter. Specifies the namespace to use for the generated class source for the strongly typed resource. If this parameter is not specified, any strongly typed resources are in the global namespace.
`TLogReadFiles`	Optional ITaskItem `[]` read-only parameter. Gets an array of items that represent the read tracking logs.
`TLogWriteFiles`	Optional ITaskItem `[]` read-only parameter. Gets an array of items that represent the write tracking logs.
`ToolArchitecture`	Optional System.String parameter. Used to determine whether or not *Tracker.exe* needs to be used to spawn *ResGen.exe*. Should be parsable to a member of the ExecutableType enumeration. If `String.Empty`, uses a heuristic to determine a default architecture. Should be parsable to a member of the Microsoft.Build.Utilities.ExecutableType enumeration.

PARAMETER	DESCRIPTION
`TrackerFrameworkPath`	Optional `String` parameter. Specifies the path to the appropriate .NET Framework location that contains *FileTracker.dll*. If set, the user takes responsibility for making sure that the bitness of the *FileTracker.dll* that they pass matches the bitness of the *ResGen.exe* that they intend to use. If not set, the task decides the appropriate location based on the current .NET Framework version.
`TrackerLogDirectory`	Optional `String` parameter. Specifies the intermediate directory into which the tracking logs from running this task will be placed.
`TrackerSdkPath`	Optional `String` parameter. Specifies the path to the appropriate Windows SDK location that contains *Tracker.exe*. If set, the user takes responsibility for making sure that the bitness of the *Tracker.exe* that they pass matches the bitness of the *ResGen.exe* that they intend to use. If not set, the task decides the appropriate location based on the current Windows SDK.
`TrackFileAccess`	Optional Boolean parameter. If true, the directory of the input file is used for resolving relative file paths.
`UseSourcePath`	Optional `Boolean` parameter. If `true`, specifies that the input file's directory is to be used for resolving relative file paths.

Remarks

Because *.resx* files may contain links to other resource files, it is not sufficient to simply compare *.resx* and *.resources* file timestamps to see if the outputs are up-to-date. Instead, the `GenerateResource` task follows the links in the *.resx* files and checks the timestamps of the linked files as well. This means that you should not generally use `Inputs` and `Outputs` attributes on the target containing the `GenerateResource` task, as this may cause it to be skipped when it should actually run.

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

When using MSBuild 4.0 to target .NET 3.5 projects, the build may fail on x86 resources. To work around this problem, you can build the target as an AnyCPU assembly.

Example

The following example uses the `GenerateResource` task to generate *.resources* files from the files specified by the

`Resx` item collection.

```
<GenerateResource
    Sources="@(Resx)"
    OutputResources="@(Resx->'$(IntermediateOutputPath)%(Identity).resources')">
    <Output
        TaskParameter="OutputResources"
        ItemName="Resources"/>
</GenerateResource>
```

The `GenerateResource` task uses the <LogicalName> metadata of an <EmbeddedResource> item to name the resource that is embedded in an assembly.

Assuming that the assembly is named myAssembly, the following code generates an embedded resource named *someQualifier.someResource.resources*:

```
<ItemGroup>
    <EmbeddedResource Include="myResource.resx">
        <LogicalName>someQualifier.someResource.resources</LogicalName>
    </EmbeddedResource>
</ItemGroup>
```

Without the <LogicalName> metadata, the resource would be named *myAssembly.myResource.resources*. This example applies only to the Visual Basic and Visual C# build process.

See also

- Tasks
- Task reference

GenerateTrustInfo task

2/21/2019 • 2 minutes to read • Edit Online

Generates the application trust from the base manifest, and from the `TargetZone` and `ExcludedPermissions` parameters.

Parameters

The following table describes the parameters of the `GenerateTrustInfo` task.

PARAMETER	DESCRIPTION
`ApplicationDependencies`	Optional ITaskItem `[]` parameter. Specifies the dependent assemblies.
`BaseManifest`	Optional ITaskItem parameter. Specifies the base manifest to generate the application trust from.
`ExcludedPermissions`	Optional `String` parameter. Specifies one or more semicolon-separated permission identity values to be excluded from the zone default permission set.
`TargetZone`	Optional `String` parameter. Specifies a zone default permission set, which is obtained from machine policy.
`TrustInfoFile`	Required ITaskItem output parameter. Specifies the file that contains the application security trust information.

Remarks

In addition to having the parameters that are listed in the table, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

See also

- Tasks
- Task reference

GetAssemblyIdentity task

2/21/2019 • 2 minutes to read • Edit Online

Retrieves the assembly identities from the specified files and outputs the identity information.

Task parameters

The following table describes the parameters of the `GetAssemblyIdentity` task.

PARAMETER	DESCRIPTION
`Assemblies`	Optional ITaskItem `[]` output parameter. Contains the retrieved assembly identities.
`AssemblyFiles`	Required ITaskItem `[]` parameter. Specifies the files to retrieve identities from.

Remarks

The items output by the `Assemblies` parameter contain item metadata entries named `Version`, `PublicKeyToken`, and `Culture`.

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following example retrieves the identity of the files specified in the `MyAssemblies` item, and outputs them into the `MyAssemblyIdentities` item.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <ItemGroup>
        <MyAssemblies Include="File1.dll;File2.dll" />
    </ItemGroup>
    <Target Name="RetrieveIdentities">
        <GetAssemblyIdentity AssemblyFiles="@(MyAssemblies)">
            <Output TaskParameter="Assemblies" ItemName="MyAssemblyIdentities" />
        </GetAssemblyIdentity>
    </Target>
</Project>
```

See also

- Tasks
- Task reference

GetFileHash task

2/21/2019 • 2 minutes to read • Edit Online

Computes checksums of the contents of a file or set of files.

This task was added in 15.8, but requires a workaround to use for MSBuild versions below 16.0.

Task parameters

The following table describes the parameters of the `GetFileHash` task.

PARAMETER	DESCRIPTION
`Files`	Required ITaskItem `[]` parameter. The files to be hashed.
`Items`	ITaskItem `[]` output parameter. The `Files` input with additional metadata set to the file hash.
`Hash`	`String` output parameter. The hash of the file. This output is only set if there was exactly one item passed in.
`Algorithm`	Optional `String` parameter. The algorithm. Allowed values: `SHA256`, `SHA384`, `SHA512`. Default = `SHA256`.
`MetadataName`	Optional `String` parameter. The metadata name where the hash is stored in each item. Defaults to `FileHash`.
`HashEncoding`	Optional `String` parameter. The encoding to use for generated hashes. Defaults to `hex`. Allowed values = `hex`, `base64`.

Example

The following example uses the `GetFileHash` task to determine and print the checksum of the `FilesToHash` items.

```
<Project>
  <ItemGroup>
    <FilesToHash Include="$(MSBuildThisFileDirectory)\*" />
  </ItemGroup>
  <Target Name="GetHash">
    <GetFileHash Files="@(FilesToHash)">
      <Output
          TaskParameter="Items"
          ItemName="FilesWithHashes" />
    </GetFileHash>

    <Message Importance="High"
             Text="@(FilesWithHashes->'%(Identity): %(FileHash)')" />
  </Target>
</Project>
```

See also

- Tasks
- Task reference

GetFrameworkPath task

6/6/2019 • 2 minutes to read • Edit Online

Retrieves the path to the .NET Framework assemblies.

Task parameters

The following table describes the parameters of the `GetFrameworkPath` task.

PARAMETER	DESCRIPTION
`FrameworkVersion11Path`	Optional `String` output parameter. Contains the path to the framework version 1.1 assemblies, if present. Otherwise returns `null`.
`FrameworkVersion20Path`	Optional `String` output parameter. Contains the path to the framework version 2.0 assemblies, if present. Otherwise returns `null`.
`FrameworkVersion30Path`	Optional `String` output parameter. Contains the path to the framework version 3.0 assemblies, if present. Otherwise returns `null`.
`FrameworkVersion35Path`	Optional `String` output parameter. Contains the path to the framework version 3.5 assemblies, if present. Otherwise returns `null`.
`FrameworkVersion40Path`	Optional `String` output parameter. Contains the path to the framework version 4.0 assemblies, if present. Otherwise returns `null`.
`Path`	Optional `String` output parameter. Contains the path to the latest framework assemblies, if any are available. Otherwise returns `null`.

Remarks

If several versions of the .NET Framework are installed, this task returns the version that MSBuild is designed to run on.

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following example uses the `GetFrameworkPath` task to store the path to the .NET Framework in the `FrameworkPath` property.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <Target Name="GetPath">
        <GetFrameworkPath>
            <Output
                TaskParameter="Path"
                PropertyName="FrameworkPath" />
        </GetFrameworkPath>
    </Target>
</Project>
```

See also

- Tasks
- Task reference

GetFrameworkSdkPath task

2/21/2019 • 2 minutes to read • Edit Online

Retrieves the path to the Windows Software Development Kit (SDK).

Task parameters

The following table describes the parameters of the `GetFrameworkSdkPath` task.

PARAMETER	DESCRIPTION
`FrameworkSdkVersion20Path`	Optional `String` read-only output parameter. Returns the path to the .NET SDK version 2.0, if present. Otherwise returns `String.Empty`.
`FrameworkSdkVersion35Path`	Optional `String` read-only output parameter. Returns the path to the .NET SDK version 3.5, if present. Otherwise returns `String.Empty`.
`FrameworkSdkVersion40Path`	Optional `String` read-only output parameter. Returns the path to the .NET SDK version 4.0, if present. Otherwise returns `String.Empty`.
`Path`	Optional `String` output parameter. Contains the path to the latest .NET SDK, if any version is present. Otherwise returns `String.Empty`.

Remarks

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following example uses the `GetFrameworkSdkPath` task to store the path to the Windows SDK in the `SdkPath` property.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <Target Name="GetPath">
        <GetFrameworkSdkPath>
            <Output
                TaskParameter="Path"
                PropertyName="SdkPath" />
        </GetFrameworkSdkPath>
        <Message Text="$(SdkPath)"/>
    </Target>
</Project>
```

See also

- Tasks
- Task reference

GetReferenceAssemblyPaths task

2/21/2019 • 2 minutes to read • Edit Online

Returns the reference assembly paths of the various frameworks.

Parameters

The following table describes the parameters of the `GetReferenceAssemblyPaths` task.

PARAMETER	DESCRIPTION
`ReferenceAssemblyPaths`	Optional `String[]` output parameter. Returns the path, based on the `TargetFrameworkMoniker` parameter. If the `TargetFrameworkMoniker` is null or empty, this path will be `String.Empty`.
`FullFrameworkReferenceAssemblyPaths`	Optional `String[]` output parameter. Returns the path, based on the `TargetFrameworkMoniker` parameter, without considering the profile part of the moniker. If the `TargetFrameworkMoniker` is null or empty, this path will be `String.Empty`.
`TargetFrameworkMoniker`	Optional `String` parameter. Specifies the target framework moniker that is associated with the reference assembly paths.
`RootPath`	Optional `String` parameter. Specifies the root path to use to generate the reference assembly path.
`BypassFrameworkInstallChecks`	Optional Boolean parameter. If `true`, bypasses the basic checks that `GetReferenceAssemblyPaths` performs by default to ensure that certain runtime frameworks are installed, depending on the target framework.
`TargetFrameworkMonikerDisplayName`	Optional `String` output parameter. Specifies the display name for the target framework moniker.

Remarks

In addition to having the parameters that are listed in the table, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

See also

- Tasks
- Task reference

LC task

2/21/2019 • 2 minutes to read • Edit Online

Wraps *LC.exe*, which generates a *.license* file from a *.licx* file. For more information on *LC.exe*, see Lc.exe (License Compiler).

Parameters

The following table describes the parameters for the `LC` task.

PARAMETER	DESCRIPTION
`LicenseTarget`	Required ITaskItem parameter. Specifies the executable for which the *.licenses* files are generated.
`NoLogo`	Optional `Boolean` parameter. Suppresses the Microsoft startup banner display.
`OutputDirectory`	Optional `String` parameter. Specifies the directory in which to place the output *.licenses* files.
`OutputLicense`	Optional ITaskItem output parameter. Specifies the name of the *.licenses* file. If you do not specify a name, the name of the *.licx* file is used and the *.licenses* file is placed in the directory that contains the *.licx* file.
`ReferencedAssemblies`	Optional ITaskItem `[]` parameter. Specifies the referenced components to load when generating the *.license* file.
`SdkToolsPath`	Optional `String` parameter. Specifies the path to the SDK tools, such as *resgen.exe*.
`Sources`	Required ITaskItem `[]` parameter. Specifies the items that contain licensed components to include in the *.licenses* file. For more information, see the documentation for the `/complist` switch in Lc.exe (License Compiler).

In addition to the parameters listed above, this task inherits parameters from the ToolTaskExtension class, which itself inherits from the ToolTask class. For a list of these additional parameters and their descriptions, see ToolTaskExtension base class.

Example

The following example uses the `LC` task to compile licenses.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
<!-- Item declarations, etc -->

    <Target Name="CompileLicenses">
        <LC
            Sources="@(LicxFile)"
            LicenseTarget="$(TargetFileName)"
            OutputDirectory="$(IntermediateOutputPath)"
            OutputLicenses="$(IntermediateOutputPath)$(TargetFileName).licenses"
            ReferencedAssemblies="@(ReferencePath);@(ReferenceDependencyPaths)">

            <Output
                TaskParameter="OutputLicenses"
                ItemName="CompiledLicenseFile"/>
        </LC>
    </Target>
</Project>
```

See also

- Tasks
- Task reference

MakeDir task

2/21/2019 • 2 minutes to read • Edit Online

Creates directories and, if necessary, any parent directories.

Parameters

The following table describes the parameters of the `MakeDir` task.

PARAMETER	DESCRIPTION
`Directories`	Required ITaskItem `[]` parameter. The set of directories to create.
`DirectoriesCreated`	Optional ITaskItem `[]` output parameter. The directories that are created by this task. If some directories could not be created, this may not contain all of the items that were passed into the `Directories` parameter.

Remarks

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following code example uses the `MakeDir` task to create the directory specified by the `OutputDirectory` property.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <PropertyGroup>
        <OutputDirectory>\Output\</OutputDirectory>
    </PropertyGroup>

    <Target Name="CreateDirectories">
        <MakeDir
            Directories="$(OutputDirectory)"/>
    </Target>

</Project>
```

See also

- Tasks
- Task reference

Message task

7/31/2019 • 2 minutes to read • Edit Online

Logs a message during a build.

Parameters

The following table describes the parameters of the `Message` task.

PARAMETER	DESCRIPTION
`Importance`	Optional `String` parameter. Specifies the importance of the message. This parameter can have a value of `high`, `normal` or `low`. The default value is `normal`.
`Text`	Optional `String` parameter. The error text to log.

Remarks

The `Message` task allows MSBuild projects to issue messages to loggers at different steps in the build process.

If the `Condition` parameter evaluates to `true`, the value of the `Text` parameter will be logged and the build will continue to execute. If a `Condition` parameter does not exist, the message text is logged. For more information on logging, see Obtain build logs.

By default, the message is sent to the MSBuild console logger. This can be changed by setting the Log parameter. The logger interprets the `Importance` parameter. Typically, a message set to `high` is sent when logger verbosity is set to LoggerVerbosity `Minimal` or higher. A message set to `low` is sent when logger verbosity is set to LoggerVerbosity `Detailed`.

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following code example logs messages to all registered loggers.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <Target Name="DisplayMessages">
        <Message Text="Project File Name = $(MSBuildProjectFile)" />
        <Message Text="Project Extension = $(MSBuildProjectExtension)" />
    </Target>
    ...
</Project>
```

See also

- Task reference
- Obtain build logs

Move task

2/21/2019 • 2 minutes to read • Edit Online

Moves files to a new location.

Parameters

The folowing table describes the parameters of the `Move` task.

PARAMETER	DESCRIPTION
`DestinationFiles`	Optional ITaskItem `[]` output parameter. Specifies the list of files to move the source files to. This list is expected to be a one-to-one mapping to the list that is specified in the `SourceFiles` parameter. That is, the first file specified in `SourceFiles` will be moved to the first location specified in `DestinationFiles`, and so forth.
`DestinationFolder`	Optional ITaskItem parameter. Specifies the directory to which you want to move the files.
`MovedFiles`	Optional ITaskItem `[]` output parameter. Contains the items that were successfully moved.
`OverwriteReadOnlyFiles`	Optional `Boolean` parameter. If `true`, overwrites files even if they are marked as read-only files.
`SourceFiles`	Required ITaskItem `[]` parameter. Specifies the files to move.

Remarks

Either the `DestinationFolder` parameter or the `DestinationFiles` parameter must be specified, but not both. If both are specified, the task fails and an error is logged.

The `Move` task creates folders as required for the desired destination files.

In addition to having the parameters that are listed in the table, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

See also

- Tasks
- Task reference

MSBuild task

10/2/2019 • 6 minutes to read • Edit Online

Builds MSBuild projects from another MSBuild project.

Parameters

The following table describes the parameters of the `MSBuild` task.

PARAMETER	DESCRIPTION
`BuildInParallel`	Optional `Boolean` parameter. If `true`, the projects specified in the `Projects` parameter are built in parallel if it is possible. Default is `false`.
`Projects`	Required ITaskItem `[]` parameter. Specifies the project files to build.
`Properties`	Optional `String` parameter. A semicolon-delimited list of property name/value pairs to apply as global properties to the child project. When you specify this parameter, it is functionally equivalent to setting properties that have the **-property** switch when you build with *MSBuild.exe*. For example: `Properties="Configuration=Debug;Optimize=$(Optimize)"` When you pass properties to the project through the `Properties` parameter, MSBuild might create a new instance of the project even if the project file has already been loaded. MSBuild creates a single project instance for a given project path and a unique set of global properties. For example, this behavior allows you to create multiple MSBuild tasks that call *myproject.proj*, with Configuration=Release and you get a single instance of *myproject.proj* (if no unique properties are specified in the task). If you specify a property that has not yet been seen by MSBuild, MSBuild creates a new instance of the project, which can be built in parallel to other instances of the project. For example, a Release configuration can build at the same time as a Debug configuration.
`RebaseOutputs`	Optional `Boolean` parameter. If `true`, the relative paths of target output items from the built projects have their paths adjusted to be relative to the calling project. Default is `false`.
`RemoveProperties`	Optional `String` parameter. Specifies the set of global properties to remove.

PARAMETER	DESCRIPTION
`RunEachTargetSeparately`	Optional `Boolean` parameter. If `true`, the MSBuild task invokes each target in the list passed to MSBuild one at a time, instead of at the same time. Setting this parameter to `true` guarantees that subsequent targets are invoked even if previously invoked targets failed. Otherwise, a build error would stop invocation of all subsequent targets. Default is `false`.
`SkipNonexistentProjects`	Optional `Boolean` parameter. If `true`, project files that do not exist on the disk will be skipped. Otherwise, such projects will cause an error.
`StopOnFirstFailure`	Optional `Boolean` parameter. If `true`, when one of the projects fails to build, no more projects will be built. Currently this is not supported when building in parallel (with multiple processors).
`TargetAndPropertyListSeparators`	Optional `String[]` parameter. Specifies a list of targets and properties as `Project` item metadata). Separators will be un-escaped before processing. e.g. %3B (an escaped ';') will be treated as if it were an un-escaped ';'.
`TargetOutputs`	Optional ITaskItem `[]` read-only output parameter. Returns the outputs of the built targets from all the project files. Only the outputs from the targets that were specified are returned, not any outputs that may exist on targets that those targets depend on. The `TargetOutputs` parameter also contains the following metadata: - `MSBuildSourceProjectFile`: The MSBuild project file that contains the target that set the outputs. - `MSBuildSourceTargetName`: The target that set the outputs. **Note:** If you want to identify the outputs from each project file or target separately, run the `MSBuild` task separately for each project file or target. If you run the `MSBuild` task only once to build all the project files, the outputs of all the targets are collected into one array.
`Targets`	Optional `String` parameter. Specifies the target or targets to build in the project files. Use a semicolon to separate a list of target names. If no targets are specified in the `MSBuild` task, the default targets specified in the project files are built. **Note:** The targets must occur in all the project files. If they do not, a build error occurs.

PARAMETER	DESCRIPTION
`ToolsVersion`	Optional `String` parameter. Specifies the `ToolsVersion` to use when building projects passed to this task. Enables an MSBuild task to build a project that targets a different version of the .NET Framework than the one specified in the project. Valid values are `2.0`, `3.0` and `3.5`. Default value is `3.5`.

Remarks

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Unlike using the Exec task to start *MSBuild.exe*, this task uses the same MSBuild process to build the child projects. The list of already-built targets that can be skipped is shared between the parent and child builds. This task is also faster because no new MSBuild process is created.

This task can process not only project files but also solution files.

Any configuration that is required by MSBuild to enable projects to build at the same time, even if the configuration involves remote infrastructure (for example, ports, protocols, timeouts, retries, and so forth), must be made configurable by using a configuration file. When possible, configuration items should be able to be specified as task parameters on the `MSBuild` task.

Beginning in MSBuild 3.5, Solution projects now surface TargetOutputs from all of the sub-projects it builds.

Pass properties to projects

In versions of MSBuild prior to MSBuild 3.5, passing different sets of properties to different projects listed in the MSBuild item was challenging. If you used the Properties attribute of the MSBuild task, then its setting was applied to all of the projects being built unless you batched the MSBuild task and conditionally provided different properties for each project in the item list.

MSBuild 3.5, however, provides two new reserved metadata items, Properties and AdditionalProperties, that provide you a flexible way to pass different properties for different projects being built using the MSBuild task.

> **NOTE**
> These new metadata items are applicable only to items passed in the Projects attribute of the MSBuild task.

Multi-processor build benefits

One of the major benefits of using this new metadata occurs when you build your projects in parallel on a multi-processor system. The metadata allows you to consolidate all projects into a single MSBuild task call without having to perform any batching or conditional MSBuild tasks. And when you call only a single MSBuild task, all of the projects listed in the Projects attribute will be built in parallel. (Only, however, if the `BuildInParallel=true` attribute is present in the MSBuild task.) For more information, see Build multiple projects in parallel.

Properties metadata

When specified, Properties metadata overrides the task's Properties parameter, while AdditionalProperties metadata gets appended to the parameter's definitions.

A common scenario is when you are building multiple solution files using the MSBuild task, only using different build configurations. You may want to build solution a1 using the Debug configuration and solution a2 using the Release configuration. In MSBuild 2.0, this project file would look like the following:

> **NOTE**
> In the following example, "..." represents additional solution files.

a.proj

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <Target Name="Build">
        <MSBuild Projects="a1.sln..." Properties="Configuration=Debug"/>
        <MSBuild Projects="a2.sln" Properties="Configuration=Release"/>
    </Target>
</Project>
```

By using the Properties metadata, however, you can simplify this to use a single MSBuild task, as shown by the following:

a.proj

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <ItemGroup>
        <ProjectToBuild Include="a1.sln...">
            <Properties>Configuration=Debug</Properties>
        </ProjectToBuild>
        <ProjectToBuild Include="a2.sln">
            <Properties>Configuration=Release</Properties>
        </ProjectToBuild>
    </ItemGroup>
    <Target Name="Build">
        <MSBuild Projects="@(ProjectToBuild)"/>
    </Target>
</Project>
```

- or -

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <ItemGroup>
        <ProjectToBuild Include="a1.sln..."/>
        <ProjectToBuild Include="a2.sln">
            <Properties>Configuration=Release</Properties>
        </ProjectToBuild>
    </ItemGroup>
    <Target Name="Build">
        <MSBuild Projects="@(ProjectToBuild)"
          Properties="Configuration=Debug"/>
    </Target>
</Project>
```

AdditionalProperties metadata

Consider the following scenario where you are building two solution files using the MSBuild task, both using the Release configuration, but one using the x86 architecture and the other using the ia64 architecture. In

MSBuild 2.0, you would need to create multiple instances of the MSBuild task: one to build the project using the Release configuration with the x86 Architecture, the other using the Release configuration with the ia64 architecture. Your project file would look like the following:

a.proj

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <Target Name="Build">
        <MSBuild Projects="a1.sln..." Properties="Configuration=Release;
          Architecture=x86"/>
        <MSBuild Projects="a2.sln" Properties="Configuration=Release;
          Architecture=ia64"/>
    </Target>
</Project>
```

By using the AdditionalProperties metadata, you can simplify this to use a single MSBuild task by using the following:

a.proj

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <ItemGroup>
        <ProjectToBuild Include="a1.sln...">
            <AdditionalProperties>Architecture=x86
              </AdditionalProperties>
        </ProjectToBuild>
        <ProjectToBuild Include="a2.sln">
            <AdditionalProperties>Architecture=ia64
              </AdditionalProperties>
        </ProjectToBuild>
    </ItemGroup>
    <Target Name="Build">
        <MSBuild Projects="@(ProjectToBuild)"
          Properties="Configuration=Release"/>
    </Target>
</Project>
```

Example

The following example uses the `MSBuild` task to build the projects specified by the `ProjectReferences` item collection. The resulting target outputs are stored in the `AssembliesBuiltByChildProjects` item collection.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <ItemGroup>
        <ProjectReferences Include="*.*proj" />
    </ItemGroup>

    <Target Name="BuildOtherProjects">
        <MSBuild
            Projects="@(ProjectReferences)"
            Targets="Build">
            <Output
                TaskParameter="TargetOutputs"
                ItemName="AssembliesBuiltByChildProjects" />
        </MSBuild>
    </Target>

</Project>
```

See also

- Tasks
- Task reference

ReadLinesFromFile task

2/21/2019 • 2 minutes to read • Edit Online

Reads a list of items from a text file.

Parameters

The following table describes the parameters of the `ReadLinesFromFile` task.

PARAMETER	DESCRIPTION
`File`	Required ITaskItem parameter. Specifies the file to read. The file must have one item on each line.
`Lines`	Optional ITaskItem `[]` output parameter. Contains the lines read from the file.

Remarks

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following example uses the `ReadLinesFromFile` task to create items from a list in a text file. The items read from the file are stored in the `ItemsFromFile` item collection.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <ItemGroup>
        <MyTextFile Include="Items.txt"/>
    </ItemGroup>

    <Target Name="ReadFromFile">
        <ReadLinesFromFile
            File="@(MyTextFile)" >
            <Output
                TaskParameter="Lines"
                ItemName="ItemsFromFile"/>
        </ReadLinesFromFile>
    </Target>

</Project>
```

See also

- Task reference
- MSBuild concepts

- Tasks

RegisterAssembly task

6/6/2019 • 2 minutes to read • Edit Online

Reads the metadata within the specified assembly and adds the necessary entries to the registry, which allows COM clients to create .NET Framework classes transparently. The behavior of this task is similar, but not identical, to that of the Regasm.exe (Assembly Registration tool).

Parameters

The following table describes the parameters of the `RegisterAssembly` task.

PARAMETER	DESCRIPTION
`Assemblies`	Required ITaskItem `[]` parameter. Specifies the assemblies to be registered with COM.
`AssemblyListFile`	Optional ITaskItem parameter. Contains information about the state between the `RegisterAssembly` task and the UnregisterAssembly task. This information prevents the `UnregisterAssembly` task from attempting to unregister an assembly that failed to register in the `RegisterAssembly` task.
`CreateCodeBase`	Optional `Boolean` parameter. If `true`, creates a codebase entry in the registry, which specifies the file path for an assembly that is not installed in the global assembly cache. You should not specify this option if you will subsequently install the assembly that you are registering into the global assembly cache.
`TypeLibFiles`	Optional ITaskItem `[]` output parameter. Specifies the type library to generate from the specified assembly. The generated type library contains definitions of the accessible types defined within the assembly. The type library is only generated if one of the following conditions is true: - A type library of that name does not exist at that location. - A type library exists but it's older than the assembly being passed in. If the type library is newer than the assembly being passed, a new one won't be created, but the assembly will still be registered. If this parameter is specified, it must have the same number of items as the `Assemblies` parameter or the task will fail. If no inputs are specified, the task will default to the name of the assembly and change the extension of the item to *.tlb*.

Remarks

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following example uses the `RegisterAssembly` task to register the assembly specified by the `MyAssemblies` item collection.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <ItemGroup>
        <MyAssemblies Include="MyAssembly.dll" />
    <ItemGroup>

    <Target Name="RegisterAssemblies">
        <RegisterAssembly
            Assemblies="@(MyAssemblies)" >
    </Target>

</Project>
```

See also

- Tasks
- Task reference

RemoveDir task

2/21/2019 • 2 minutes to read • Edit Online

Removes the specified directories and all of its files and subdirectories.

Parameters

The following table describes the parameters of the `RemoveDir` task.

PARAMETER	DESCRIPTION
`Directories`	Required ITaskItem `[]` parameter. Specifies the directories to delete.
`RemovedDirectories`	Optional ITaskItem `[]` output parameter. Contains the directories that were successfully deleted.

Remarks

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following example removes the directories specified by the `OutputDirectory` and `DebugDirectory` properties. These paths are treated as relative to the project directory.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2005">

    <PropertyGroup>
        <OutputDirectory>\Output\</OutputDirectory>
        <DebugDirectory>\Debug\</DebugDirectory>
    </PropertyGroup>

    <Target Name="RemoveDirectories">
        <RemoveDir
            Directories="$(OutputDirectory);$(DebugDirectory)" />
    </Target>

</Project>
```

See also

- Tasks
- Task reference

RemoveDuplicates task

2/21/2019 • 2 minutes to read • Edit Online

Removes duplicate items from the specified item collection.

Parameters

The following table describes the parameters of the `RemoveDuplicates` task.

PARAMETER	DESCRIPTION
`Filtered`	Optional ITaskItem `[]` output parameter. Contains an item collection with all duplicate items removed. The order of the input items is preserved, keeping the first instance of each duplicate item.
`Inputs`	Optional ITaskItem `[]` parameter. The item collection to remove duplicate items from.

Remarks

This task is case insensitive and does not compare item metadata when determining duplicates.

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following example uses the `RemoveDuplicates` task to remove duplicate items from the `MyItems` item collection. When the task is complete, the `FilteredItems` item collection contains one item.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <ItemGroup>
        <MyItems Include="MyFile.cs"/>
        <MyItems Include="MyFile.cs">
            <Culture>fr</Culture>
        </MyItems>
        <MyItems Include="myfile.cs"/>
    </ItemGroup>

    <Target Name="RemoveDuplicateItems">
        <RemoveDuplicates
            Inputs="@(MyItems)">
            <Output
                TaskParameter="Filtered"
                ItemName="FilteredItems"/>
        </RemoveDuplicates>
    </Target>
</Project>
```

The following example shows that the `RemoveDuplicates` task preserves its input order. When the task is complete, the `FilteredItems` item collection contains the items *MyFile2.cs*, *MyFile1.cs*, and *MyFile3.cs* in that order.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <ItemGroup>
        <MyItems Include="MyFile2.cs"/>
        <MyItems Include="MyFile1.cs" />
        <MyItems Include="MyFile3.cs" />
        <MyItems Include="myfile1.cs"/>
    </ItemGroup>

    <Target Name="RemoveDuplicateItems">
        <RemoveDuplicates
            Inputs="@(MyItems)">
            <Output
                TaskParameter="Filtered"
                ItemName="FilteredItems"/>
        </RemoveDuplicates>
    </Target>
</Project>
```

See also

- Task reference
- MSBuild concepts
- Tasks

RequiresFramework35SP1Assembly task

2/21/2019 • 2 minutes to read • Edit Online

Determines whether the application requires the .NET Framework 3.5 SP1.

Parameters

The following table describes the parameters of the `RequiresFramework35SP1Assembly` task.

PARAMETER	DESCRIPTION
`Assemblies`	Optional ITaskItem `[]` parameter. Specifies the assemblies that are referenced in the application.
`CreateDesktopShortcut`	Optional `Boolean` parameter. If `true`, creates a shortcut icon on the desktop during installation.
`DeploymentManifestEntryPoint`	Optional ITaskItem parameter. Specifies the manifest file name for the application.
`EntryPoint`	Optional ITaskItem parameter. Specifies the assembly that should be executed when the application is run.
`ErrorReportUrl`	Optional `String` parameter. Specifies the Web site that is displayed in dialog boxes that are encountered during ClickOnce installations.
`Files`	Optional ITaskItem `[]` parameter. Specifies the list of files that will be deployed when the application is published.
`ReferencedAssemblies`	Optional ITaskItem `[]` parameter. Specifies the assemblies that are referenced in the project.
`RequiresMinimumFramework35SP1`	Optional `Boolean` output parameter. If `true`, the application requires the .NET Framework 3.5 SP1.
`SigningManifests`	Optional `Boolean` output parameter. If `true`, the ClickOnce manifests are signed.

PARAMETER	DESCRIPTION
`SuiteName`	Optional `String` parameter. Specifies the name of the folder on the **Start** menu in which the application will be installed.
`TargetFrameworkVersion`	Optional `String` parameter. Specifies the version of the .NET Framework that this application targets.

Remarks

In addition to having the parameters that are listed in the table, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

See also

- Tasks
- Task reference

ResolveAssemblyReference task

4/18/2019 • 13 minutes to read • Edit Online

Determines all assemblies that depend on the specified assemblies, including second and `n` th-order dependencies.

Parameters

The following table describes the parameters of the `ResolveAssemblyReference` task.

PARAMETER	DESCRIPTION
`AllowedAssemblyExtensions`	Optional `String[]` parameter. The assembly file name extensions to use when resolving references. The default file name extensions are *.exe* and *.dll*.
`AllowedRelatedFileExtensions`	Optional `String[]` parameter. The file name extensions to use for a search for files that are related to one another. The default extensions are *.pdb* and *.xml*.
`AppConfigFile`	Optional `String` parameter. Specifies an *app.config* file from which to parse and extract bindingRedirect mappings. If this parameter is specified, the `AutoUnify` parameter must be `false`.

PARAMETER	DESCRIPTION
`AutoUnify`	Optional `Boolean` parameter. This parameter is used for building assemblies, such as DLLs, which cannot have a normal *App.Config* file. When `true`, the resulting dependency graph is automatically treated as if there were an *App.Config* file passed in to the AppConfigFile parameter. This virtual *App.Config* file has a bindingRedirect entry for each conflicting set of assemblies such that the highest version assembly is chosen. A consequence of this is that there will never be a warning about conflicting assemblies, because every conflict will have been resolved. When `true`, each distinct remapping will result in a high priority comment showing the old and new versions and that `AutoUnify` was `true`. When `true`, the AppConfigFile parameter must be empty. When `false`, no assembly version remapping will occur automatically. When two versions of an assembly are present, a warning is issued. When `false`, each distinct conflict between different versions of the same assembly results in a high-priority comment. These comments are followed by a single warning. The warning has a unique error code and contains text that reads "Found conflicts between different versions of reference and dependent assemblies".

PARAMETER	DESCRIPTION
`Assemblies`	Optional ITaskItem `[]` parameter. Specifies the items for which full paths and dependencies must be identified. These items can have either simple names like "System" or strong names like "System, Version=2.0.3500.0, Culture=neutral, PublicKeyToken=b77a5c561934e089". Items passed to this parameter may optionally have the following item metadata: - `Private` : `Boolean` value. If `true` , then the item is copied locally. The default value is `true` . - `HintPath` : `String` value. Specifies the path and file name to use as a reference. This metadata is used when {HintPathFromItem} is specified in the `SearchPaths` parameter. The default value is an empty string. - `SpecificVersion` : `Boolean` value. If `true` , then the exact name specified in the `Include` attribute must match. If `false` , then any assembly with the same simple name will work. If `SpecificVersion` is not specified, then the task examines the value in the `Include` attribute of the item. If the attribute is a simple name, it behaves as if `SpecificVersion` was `false` . If the attribute is a strong name, it behaves as if `SpecificVersion` was `true` . When used with a Reference item type, the `Include` attribute needs to be the full fusion name of the assembly to be resolved. The assembly is only resolved if fusion exactly matches the `Include` attribute. When a project targets a .NET Framework version and references an assembly compiled for a higher .NET Framework version, the reference resolves only if it has `SpecificVersion` set to `true` . When a project targets a profile and references an assembly that is not in the profile, the reference resolves only if it has `SpecificVersion` set to `true` . - `ExecutableExtension` : `String` value. When present, the resolved assembly must have this extension. When absent, *.dll* is considered first, followed by *.exe*, for each examined directory. - `SubType` : `String` value. Only items with empty SubType metadata will be resolved into full assembly paths. Items with non-empty SubType metadata are ignored. - `AssemblyFolderKey` : `String` value. This metadata is supported for legacy purposes. It specifies a user-defined registry key, such as **hklm\<VendorFolder>**, that `Assemblies` should use to resolve assembly references.

PARAMETER	DESCRIPTION
`AssemblyFiles`	Optional ITaskItem `[]` parameter. Specifies a list of fully qualified assemblies for which to find dependencies. Items passed to this parameter may optionally have the following item metadata: - `Private`: an optional `Boolean` value. If true, the item is copied locally. - `FusionName`: optional `String` metadata. Specifies the simple or strong name for this item. If this attribute is present, it can save time because the assembly file does not have to be opened to get the name.
`AutoUnify`	Optional `Boolean` parameter. If `true`, the resulting dependency graph is automatically treated as if there were an *App.Config* file passed in to the AppConfigFile parameter. This virtual *App.Config* file has a bindingRedirect entry for each conflicting set of assemblies so that the highest version assembly is chosen. A result of this is that there will never be a warning about conflicting assemblies because every conflict will have been resolved. Each distinct remapping will cause a high priority comment that indicates the old and new versions and the fact that this was done automatically because `AutoUnify` was `true`. If `false`, no assembly version remapping will occur automatically. When two versions of an assembly are present, there will be a warning. Each distinct conflict between different versions of the same assembly will cause a high priority comment. After all these comments are displayed, there will be a single warning with a unique error code and text that reads "Found conflicts between different versions of reference and dependent assemblies". The default value is `false`.
`CandidateAssemblyFiles`	Optional `String[]` parameter. Specifies a list of assemblies to use for the search and resolution process. Values passed to this parameter must be absolute file names or project-relative file names. Assemblies in this list will be considered when the `SearchPaths` parameter contains {CandidateAssemblyFiles} as one of the paths to consider.

PARAMETER	DESCRIPTION
`CopyLocalDependenciesWhenParentReferenceInGac`	Optional Boolean parameter. If true, to determine if a dependency should be copied locally, one of the checks done is to see if the parent reference in the project file has the Private metadata set. If set, then the Private value is used as a dependency. If the metadata is not set, then the dependency goes through the same checks as the parent reference. One of these checks is to see if the reference is in the GAC. If a reference is in the GAC, then it is not copied locally, because it is assumed to be in the GAC on the target machine. This only applies to a specific reference and not its dependencies. For example, a reference in the project file that is in the GAC is not copied locally, but its dependencies are copied locally because they are not in the GAC. If false, project file references are checked to see if they are in the GAC, and are copied locally as appropriate. Dependencies are checked to see if they are in the GAC and are also checked to see if the parent reference from the project file is in the GAC. If the parent reference from the project file is in the GAC, the dependency is not copied locally. Whether this parameter is true or false, if there are multiple parent references and any of them are not in the GAC, then all of them are copied locally.
`CopyLocalFiles`	Optional ITaskItem `[]` read-only output parameter. Returns every file in the `ResolvedFiles`, `ResolvedDependencyFiles`, `RelatedFiles`, `SatelliteFiles`, and `ScatterFiles` parameters that has `CopyLocal` item metadata with a value of `true`.
`FilesWritten`	Optional ITaskItem `[]` output parameter. Contains the items written to disk.
`FindDependencies`	Optional `Boolean` parameter. If `true`, dependencies will be found. Otherwise, only primary references are found. The default value is `true`.
`FindRelatedFiles`	Optional `Boolean` parameter. If `true`, related files such as *.pdb* files and *xml* files will be found. The default value is `true`.
`FindSatellites`	Optional `Boolean` parameter. If `true`, satellite assemblies will be found. The default value is `true.`

PARAMETER	DESCRIPTION
`FindSerializationAssemblies`	Optional `Boolean` parameter. If `true`, then the task searches for serialization assemblies. The default value is `true`.
`FullFrameworkAssemblyTables`	Optional ITaskItem `[]` parameter. Specifies items that have "FrameworkDirectory" metadata to associate a redist list with a particular framework directory. If the association is not made, an error will be logged. The resolve assembly reference (RAR) logic uses the target framework directory if a FrameworkDirectory is not set.
`FullFrameworkFolders`	Optional System.String `[]` parameter. Specifies the folders that contain a RedistList directory. This directory represents the full framework for a given client profile, for example, *%programfiles%\reference assemblies\microsoft\framework\v4.0*.
`FullTargetFrameworkSubsetNames`	Optional `String[]` parameter. Contains a list of target framework subset names. If a subset name in the list matches one in the `TargetFrameworkSubset` name property, then the system excludes that particular target framework subset at build time.
`IgnoreDefaultInstalledAssemblyTables`	Optional `Boolean` parameter. If `true`, then the task searches for and uses additional installed assembly tables (or, "Redist Lists") that are found in the *\RedistList* directory under `TargetFrameworkDirectories`. The default value is `false.`
`IgnoreDefaultInstalledAssemblySubsetTables`	Optional `Boolean` parameter. If `true`, then the task searches for and uses additional installed assembly subset tables (or, "Subset Lists") that are found in the *\SubsetList* directory under `TargetFrameworkDirectories`. The default value is `false.`
`InstalledAssemblySubsetTables`	Optional ITaskItem `[]` parameter. Contains a list of XML files that specify the assemblies that are expected to be in the target subset. As an option, items in this list can specify the "FrameworkDirectory" metadata to associate an `InstalledAssemblySubsetTable` with a particular framework directory. If there is only one `TargetFrameworkDirectories` element, then any items in this list that lack the "FrameworkDirectory" metadata are treated as though they are set to the unique value that is passed to `TargetFrameworkDirectories`.

PARAMETER	DESCRIPTION
`InstalledAssemblyTables`	Optional `String` parameter. Contains a list of XML files that specify the assemblies that are expected to be installed on the target computer. When `InstalledAssemblyTables` is set, earlier versions of the assemblies in the list are merged into the newer versions that are listed in the XML. Also, assemblies that have a setting of InGAC='true' are considered prerequisites and are set to CopyLocal='false' unless explicitly overridden. As an option, items in this list can specify "FrameworkDirectory" metadata to associate an `InstalledAssemblyTable` with a particular framework directory. However, this setting is ignored unless the Redist name begins with "Microsoft-Windows-CLRCoreComp". If there is only one `TargetFrameworkDirectories` element, then any items in this list that lack the "FrameworkDirectory" metadata are treated as if they are set to the unique value that is passed to `TargetFrameworkDirectories`.
`LatestTargetFrameworkDirectories`	Optional `String[]` parameter. Specifies a list of directories that contain the redist lists for the most current framework that can be targeted on the machine. If this is not set, then the highest framework installed on the machine for a given target framework identifier is used.
`ProfileName`	Optional `String` parameter. - Specifies the name of the framework profile to be targeted. For example, Client, Web, or Network.
`RelatedFiles`	Optional ITaskItem `[]` read-only output parameter. Contains related files, such as XML and *.pdb* files that have the same base name as a reference. The files listed in this parameter may optionally contain the following item metadata: - `Primary`: `Boolean` value. If `true`, then the file item was passed into the array by using the `Assemblies` parameter. Default value is `false`. - `CopyLocal`: `Boolean` value. Indicates whether the given reference should be copied to the output directory.

PARAMETER	DESCRIPTION
`ResolvedDependencyFiles`	Optional ITaskItem `[]` read-only output parameter. Contains the *n*th order paths to dependencies. This parameter does not include first order primary references, which are contained in the `ResolvedFiles` parameter. The items in this parameter optionally contain the following item metadata: - `CopyLocal` : `Boolean` value. Indicates whether the given reference should be copied to the output directory. - `FusionName` : `String` value. Specifies the name for this dependency. - `ResolvedFrom` : `String` value. Specifies the literal search path that this file was resolved from.
`ResolvedFiles`	Optional ITaskItem `[]` read-only output parameter. Contains a list of all primary references resolved to full paths. The items in this parameter optionally contain the following item metadata: - `CopyLocal` : `Boolean` value. Indicates whether the given reference should be copied to the output directory. - `FusionName` : `String` value. Specifies the name for this dependency. - `ResolvedFrom` : `String` value. Specifies the literal search path that this file was resolved from.
`SatelliteFiles`	Optional ITaskItem `[]` read-only output parameter. Specifies any satellite files found. These will be CopyLocal=true if the reference or dependency that caused this item to exist is CopyLocal=true. The items in this parameter optionally contain the following item metadata: - `CopyLocal` : `Boolean` value. Indicates whether the given reference should be copied to the output directory. This value is `true` if the reference or dependency that caused this item to exist has a `CopyLocal` value of `true` . - `DestinationSubDirectory` : `String` value. Specifies the relative destination directory to copy this item to.
`ScatterFiles`	Optional ITaskItem `[]` read-only output parameter. Contains the scatter files associated with one of the given assemblies. The items in this parameter optionally contain the following item metadata: - `CopyLocal` : `Boolean` value. Indicates whether the given reference should be copied to the output directory.

PARAMETER	DESCRIPTION
`SearchPaths`	Required `String[]` parameter. Specifies the directories or special locations that are searched to find the files on disk that represent the assemblies. The order in which the search paths are listed is important. For each assembly, the list of paths is searched from left to right. When a file that represents the assembly is found, that search stops and the search for the next assembly starts. This parameter accepts a semicolon-delimited list of values that can be either directory paths or special literal values from the list below: - `{HintPathFromItem}`: Specifies that the task will examine the `HintPath` metadata of the base item. - `{CandidateAssemblyFiles}`: Specifies that the task will examine the files passed in through the `CandidateAssemblyFiles` parameter. - `{Registry:` <AssemblyFoldersBase>, <RuntimeVersion>, <AssemblyFoldersSuffix> `}`: Specifies that the task will search in additional folders specified in the registry. <AssemblyFoldersBase>, <RuntimeVersion>, and <AssemblyFoldersSuffix> should be replaced with specific values for the registry location to be searched. The default specification in the common targets is {Registry:$(FrameworkRegistryBase), $(TargetFrameworkVersion), $(AssemblyFoldersSuffix), $(AssemblyFoldersExConditions)}. - `{AssemblyFolders}`: Specifies the task will use the Visual Studio.NET 2003 finding-assemblies-from-registry scheme. - `{GAC}`: Specifies the task will search in the Global Assembly Cache (GAC). - `{RawFileName}`: Specifies the task will consider the `Include` value of the item to be an exact path and file name.
`SerializationAssemblyFiles`	Optional ITaskItem `[]` read-only output parameter. Contains any XML serialization assemblies found. These items are marked CopyLocal=true if and only if the reference or dependency that caused this item to exist is CopyLocal=true. The `Boolean` metadata CopyLocal indicates whether the given reference should be copied to the output directory.
`Silent`	Optional `Boolean` parameter. If `true`, no messages are logged. The default value is `false`.
`StateFile`	Optional `String` parameter. Specifies a file name that indicates where to save the intermediate build state for this task.

PARAMETER	DESCRIPTION
`SuggestedRedirects`	Optional ITaskItem `[]` read-only output parameter. Contains one item for every distinct conflicting assembly identity, regardless of the value of the `AutoUnify` parameter. This includes every culture and PKT that was found that did not have a suitable bindingRedirect entry in the application configuration file. Each item optionally contains the following information: - `Include` attribute: Contains the full name of the assembly family with a Version field value of 0.0.0.0 - `MaxVersion` item metadata: Contains the maximum version number.
`TargetedRuntimeVersion`	Optional `String` parameter. Specifies the runtime version to target, for example, 2.0.57027 or v2.0.57027.
`TargetFrameworkDirectories`	Optional `String[]` parameter. Specifies the path of the target framework directory. This parameter is required to determine the CopyLocal status for resulting items. If this parameter is not specified, no resulting items will have a CopyLocal value of `true` unless they explicitly have a `Private` metadata value of `true` on their source item.
`TargetFrameworkMoniker`	Optional `String` parameter. The TargetFrameworkMoniker to monitor, if any. This is used for logging.
`TargetFrameworkMonikerDisplayName`	Optional `String` parameter. The display name of the TargetFrameworkMoniker to monitor, if any. This is used for logging.
`TargetFrameworkSubsets`	Optional `String[]` parameter. Contains a list of target framework subset names to be searched for in the target framework directories.
`TargetFrameworkVersion`	Optional `String` parameter. The project target framework version. The default value is empty, which means there is no filtering for the references based on target framework.

PARAMETER	DESCRIPTION
`TargetProcessorArchitecture`	Optional `String` parameter. The preferred target processor architecture. Used for resolving Global Assembly Cache (GAC) references. This parameter can have a value of `x86`, `IA64`, or `AMD64`. If this parameter is absent, the task first considers assemblies that match the architecture of the currently running process. If no assembly is found, the task considers assemblies in the GAC that have `ProcessorArchitecture` value of `MSIL` or no `ProcessorArchitecture` value.

Warnings

The following warnings are logged:

- `ResolveAssemblyReference.TurnOnAutoGenerateBindingRedirects`
- `ResolveAssemblyReference.SuggestedRedirects`
- `ResolveAssemblyReference.FoundConflicts`
- `ResolveAssemblyReference.AssemblyFoldersExSearchLocations`
- `ResolveAssemblyReference.UnifiedPrimaryReference`
- `ResolveAssemblyReference.PrimaryReference`
- `ResolveAssemblyReference.UnifiedDependency`
- `ResolveAssemblyReference.UnificationByAutoUnify`
- `ResolveAssemblyReference.UnificationByAppConfig`
- `ResolveAssemblyReference.UnificationByFrameworkRetarget`

Remarks

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

See also

- Tasks
- Task reference

ResolveComReference task

7/31/2019 • 4 minutes to read • Edit Online

Takes a list of one or more type library names or *.tlb* files and resolves those type libraries to locations on disk.

Parameters

The following table describes the parameters of the `ResolveCOMReference` task.

PARAMETER	DESCRIPTION
`DelaySign`	Optional `Boolean` parameter. If `true`, places the public key in the assembly. If `false`, fully signs the assembly.
`EnvironmentVariables`	Optional `String[]` parameter. Array of pairs of environment variables, separated by equal signs. These variables are passed to the spawned *tlbimp.exe* and *aximp.exe* in addition to, or selectively overriding, the regular environment block..
`ExecuteAsTool`	Optional `Boolean` parameter. If `true`, runs *tlbimp.exe* and *aximp.exe* from the appropriate target framework out-of-proc to generate the necessary wrapper assemblies. This parameter enables multi-targeting.
`IncludeVersionInInteropName`	Optional `Boolean` parameter. If `true`, the typelib version will be included in the wrapper name. The default is `false`.
`KeyContainer`	Optional `String` parameter. Specifies a container that holds a public/private key pair.
`KeyFile`	Optional `String` parameter. Specifies an item that contains a public/private key pair.
`NoClassMembers`	Optional `Boolean` parameter.
`ResolvedAssemblyReferences`	Optional ITaskItem `[]` output parameter. Specifies the resolved assembly references.

PARAMETER	DESCRIPTION
`ResolvedFiles`	Optional ITaskItem `[]` output parameter. Specifies the fully qualified files on disk that correspond to the physical locations of the type libraries that were provided as input to this task.
`ResolvedModules`	Optional ITaskItem `[]` parameter.
`SdkToolsPath`	Optional System.String parameter. If `ExecuteAsTool` is `true`, this parameter must be set to the SDK tools path for the framework version being targeted.
`StateFile`	Optional `String` parameter. Specifies the cache file for COM component timestamps. If not present, every run will regenerate all the wrappers.
`TargetFrameworkVersion`	Optional `String` parameter. Specifies the project target framework version. The default is `String.Empty`. which means there is no filtering for a reference based on the target framework.
`TargetProcessorArchitecture`	Optional `String` parameter. Specifies the preferred target processor architecture. Passed to the *tlbimp.exe*/machine flag after translation. The parameter value should be a member of ProcessorArchitecture.
`TypeLibFiles`	Optional ITaskItem `[]` parameter. Specifies the type library file path to COM references. Items included in this parameter may contain item metadata. For more information, see the section TypeLibFiles item metadata below.
`TypeLibNames`	Optional ITaskItem `[]` parameter. Specifies the type library names to resolve. Items included in this parameter must contain some item metadata. For more information, see the section TypeLibNames item metadata below.
`WrapperOutputDirectory`	Optional `String` parameter. The location on disk where the generated interop assembly is placed. If this item metadata is not specified, the task uses the absolute path of the directory where the project file is located.

TypeLibNames item metadata

The following table describes the item metadata available for items passed to the `TypeLibNames` parameter.

METADATA	DESCRIPTION
`GUID`	Required item metadata. The GUID for the type library. If this item metadata is not specified, the task fails.
`VersionMajor`	Required item metadata. The major version of the type library. If this item metadata is not specified, the task fails.
`VersionMinor`	Required item metadata. The minor version of the type library. If this item metadata is not specified, the task fails.
`EmbedInteropTypes`	Optional `Boolean` metadata. If `true`, embed the interop types from this reference directly into your assembly rather than generating an interop DLL.
`LocaleIdentifier`	Optional item metadata. The Locale Identifier (or LCID) for the type library. This is specified as a 32-bit value that identifies the human language preferred by a user, region, or application. If this item metadata is not specified, the task uses a default locale identifier of "0".
`WrapperTool`	Optional item metadata. Specifies the wrapper tool that is used to generate the assembly wrapper for this type library. If this item metadata is not specified, the task uses a default wrapper tool of "tlbimp". The available, case insensitive choices of typelibs are: - `Primary`: Use this wrapper tool when you want to use an already generated primary interop assembly for the COM component. When you use this wrapper tool, do not specify a wrapper output directory because that will cause the task to fail. - `TLBImp`: Use this wrapper tool when you want to generate an interop assembly for the COM component. - `AXImp`:Use this wrapper tool when you want to generate an interop assembly for an ActiveX Control.

TypeLibFiles item metadata

The following table describes the item metadata available for items passed to the `TypeLibFiles` parameter.

METADATA	DESCRIPTION
`EmbedInteropTypes`	Optional `Boolean` parameter. If `true`, embed the interop types from this reference directly into your assembly rather than generating an interop DLL.

METADATA	DESCRIPTION
`WrapperTool`	Optional item metadata. Specifies the wrapper tool that is used to generate the assembly wrapper for this type library. If this item metadata is not specified, the task uses a default wrapper tool of "tlbimp". The available, case insensitive choices of typelibs are: - `Primary`: Use this wrapper tool when you want to use an already generated primary interop assembly for the COM component. When you use this wrapper tool, do not specify a wrapper output directory because that will cause the task to fail. - `TLBImp`: Use this wrapper tool when you want to generate an interop assembly for the COM component. - `AXImp`: Use this wrapper tool when you want to generate an interop assembly for an ActiveX Control.

> **NOTE**
> The more information that you provide to uniquely identify a type library, the greater the possibility that the task will resolve to the correct file on disk.

Remarks

In addition to the parameters listed above, this task inherits parameters from the Task class. For a list of these additional parameters and their descriptions, see Task base class.

The COM DLL doesn't need to be registered on the machine for this task to work.

See also

- Tasks
- Task reference

ResolveKeySource task

2/21/2019 • 2 minutes to read • Edit Online

Determines the strong name key source.

Task parameters

The following table describes the parameters of the `ResolveKeySource` task.

PARAMETER	DESCRIPTION
`AutoClosePasswordPromptShow`	Optional `Int32` parameter. Gets or sets the amount of time, in seconds, to display the count down message.
`AutoClosePasswordPromptTimeout`	Optional `Int32` parameter. Gets or sets the amount of time, in seconds, to wait before closing the password prompt dialog.
`CertificateFile`	Optional `String` parameter. Gets or sets the path of the certificate file.
`CertificateThumbprint`	Optional `String` parameter. Gets or sets the certificate thumbprint.
`KeyFile`	Optional `String` parameter. Gets or sets the path of the key file.
`ResolvedKeyContainer`	Optional `String` output parameter. Gets or sets the resolved key container.
`ResolvedKeyFile`	Optional `String` output parameter. Gets or sets the resolved key file.
`ResolvedThumbprint`	Optional `String` output parameter. Gets or sets the resolved certificate thumbprint.
`ShowImportDialogDespitePreviousFailures`	Optional `Boolean` parameter. If `true`, show the import dialog despite previous failures.

PARAMETER	DESCRIPTION
`SuppressAutoClosePasswordPrompt`	Optional `Boolean` parameter. Gets or sets a Boolean value that specifies whether the password prompt dialog should not auto-close.

Remarks

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

See also

- Tasks
- Task reference

ResolveManifestFiles task

2/21/2019 • 2 minutes to read • Edit Online

Resolves the following items in the build process to files for manifest generation: built items, dependencies, satellites, content, debug symbols, and documentation.

Parameters

The following table describes the parameters of the `ResolveManifestFiles` task.

PARAMETER	DESCRIPTION
`DeploymentManifestEntryPoint`	Optional ITaskItem parameter. Specifies the name of the deployment manifest.
`EntryPoint`	Optional ITaskItem parameter. Specifies the managed assembly or ClickOnce manifest reference that is the entry point to the manifest.
`ExtraFiles`	Optional ITaskItem `[]` parameter. Specifies the extra files.
`ManagedAssemblies`	Optional ITaskItem `[]` parameter. Specifies the managed assemblies.
`NativeAssemblies`	Optional ITaskItem `[]` parameter. Specifies the native assemblies.
`OutputAssemblies`	Optional ITaskItem `[]` output parameter. Specifies the generated assemblies.
`OutputDeploymentManifestEntryPoint`	Optional ITaskItem output parameter. Specifies the output deployment manifest entry point.
`OutputEntryPoint`	Optional ITaskItem output parameter. Specifies the output entry point.
`OutputFiles`	Optional ITaskItem `[]` output parameter. Specifies the output files.
`PublishFiles`	Optional ITaskItem `[]` parameter. Specifies the publish files.

PARAMETER	DESCRIPTION
`SatelliteAssemblies`	Optional ITaskItem `[]` parameter. Specifies the satellite assemblies.
`SigningManifests`	Optional `Boolean` parameter. If `true`, the manifests are signed.
`TargetCulture`	Optional `String` parameter. Specifies the target culture for satellite assemblies.
`TargetFrameworkVersion`	Optional `String` parameter. Specifies the target .NET Framework version.

Remarks

In addition to having the parameters that are listed in the table, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

See also

- Tasks
- Task reference

ResolveNativeReference task

2/21/2019 • 2 minutes to read • Edit Online

Resolves native references. Implements the ResolveNativeReference class. This class supports the .NET Framework infrastructure, which is not intended to be used directly from your code.

Task parameters

The following table describes the parameters of the `ResolveNativeReference` task.

PARAMETER	DESCRIPTION
`AdditionalSearchPaths`	Required System.String `[]` parameter. Gets or sets the search paths for resolving assembly identities of native references.
`ContainedComComponents`	Optional ITaskItem `[]` output parameter. Gets or sets the COM components of the native assembly.
`ContainedLooseEtcFiles`	Optional ITaskItem `[]` output parameter. Gets or sets the loose *Etc* files listed in the native manifest.
`ContainedLooseTlbFiles`	Optional ITaskItem `[]` output parameter. Gets or sets the loose *.tlb* files of the native assembly.
`ContainedPrerequisiteAssemblies`	Optional ITaskItem `[]` output parameter. Gets or sets the assemblies that must be present before the manifest can be used.
`ContainedTypeLibraries`	Optional ITaskItem `[]` output parameter. Gets or sets the type libraries of the native assembly.
`ContainingReferenceFiles`	Optional ITaskItem `[]` output parameter. Gets or sets the reference files.
`NativeReferences`	Required ITaskItem `[]` parameter. Gets or sets the Win32 native assembly references.

Remarks

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

See also

- Tasks
- Task reference

ResolveNonMSBuildProjectOutput task

2/21/2019 • 2 minutes to read • Edit Online

Determines the output files for non-MSBuild project references.

Parameters

The following table describes the parameters of the `ResolveNonMSBuildProjectOutput` task.

PARAMETER	DESCRIPTION
`PreresolvedProjectOutputs`	Optional `String` parameter. Specifies an XML string that contains resolved project outputs.
`ProjectReferences`	Required ITaskItem `[]` parameter. Specifies the project references.
`ResolvedOutputPaths`	Optional ITaskItem `[]` output parameter. Contains the list of resolved reference paths (and preserves the original project reference attributes).
`UnresolvedProjectReferences`	Optional ITaskItem `[]` output parameter. Contains the list of project reference items that could not be resolved by using the preresolved list of outputs. Because Visual Studio only preresolves non-MSBuild projects, this means that project references in this list are in the MSBuild format.

Remarks

In addition to having the parameters that are listed in the table, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

See also

- Tasks
- Task reference

SGen task

9/11/2019 • 2 minutes to read • Edit Online

Creates an XML serialization assembly for types in the specified assembly. This task wraps the XML Serializer Generator tool (*Sgen.exe*). For more information, see XML Serializer Generator tool (Sgen.exe).

Parameters

The following table describes the parameters of the `SGen` task.

PARAMETER	DESCRIPTION
`BuildAssemblyName`	Required `String` parameter. The assembly to generate serialization code for.
`BuildAssemblyPath`	Required `String` parameter. The path to the assembly to generate serialization code for.
`DelaySign`	Optional `Boolean` parameter. If `true`, specifies that you only want to place the public key in the assembly. If `false`, specifies that you want a fully signed assembly. This parameter has no effect unless used with either the `KeyFile` or `KeyContainer` parameter.
`KeyContainer`	Optional `String` parameter. Specifies a container that holds a key pair. This will sign the assembly by inserting a public key into the assembly manifest. The task will then sign the final assembly with the private key.
`KeyFile`	Optional `String` parameter. Specifies a key pair or a public key to use to sign an assembly. The compiler inserts the public key in the assembly manifest and then signs the final assembly with the private key.
`Platform`	Optional `String` parameter. Gets or Sets the Compiler Platform used to generate the output assembly. This parameter can have a value of `x86`, `x64`, or `anycpu`. Default is `anycpu`.
`References`	Optional `String[]` parameter. Specifies the assemblies that are referenced by the types requiring XML serialization.

PARAMETER	DESCRIPTION
`SdkToolsPath`	Optional `String` parameter. Specifies the path to the SDK tools, such as *resgen.exe*.
`SerializationAssembly`	Optional ITaskItem `[]` output parameter. Contains the generated serialization assembly.
`SerializationAssemblyName`	Optional `String` parameter. Specifies the name of the generated serialization assembly.
`ShouldGenerateSerializer`	Required `Boolean` parameter. If `true`, the SGen task should generate a serialization assembly.
`Timeout`	Optional `Int32` parameter. Specifies the amount of time, in milliseconds, after which the task executable is terminated. The default value is `Int.MaxValue`, indicating that there is no time out period.
`ToolPath`	Optional `String` parameter. Specifies the location from where the task will load the underlying executable file (*sgen.exe*). If this parameter is not specified, the task uses the SDK installation path corresponding to the version of the framework that is running MSBuild.
`Types`	Optional `String[]` parameter. Gets or sets a list of specific Types to generate serialization code for. SGen will generate serialization code only for those types.
`UseProxyTypes`	Required `Boolean` parameter. If `true`, the SGen task generates serialization code only for the XML Web service proxy types.

Remarks

In addition to the parameters listed above, this task inherits parameters from the ToolTaskExtension class, which itself inherits from the ToolTask class. For a list of these additional parameters and their descriptions, see ToolTaskExtension base class.

See also

- Task reference
- Tasks
- MSBuild concepts

SignFile task

2/21/2019 • 2 minutes to read • Edit Online

Signs the specified file using the specified certificate.

Parameters

The following table describes the parameters of the `SignFile` task.

Note that SHA-256 certificates are allowed only on machines that have .NET 4.5 and higher.

> **WARNING**
> Starting in Visual Studio 2013 Update 3, this task has a new signature that allows you to specify the target framework version for the file. You are encouraged to use the new signature wherever possible, because the MSBuild process uses SHA-256 hashes only when the target framework is .NET 4.5 or higher. If the target framework is .NET 4.0 or below, the SHA-256 hash will not be used.

PARAMETER	DESCRIPTION
`CertificateThumbprint`	Required `String` parameter. Specifies the certificate to use for signing. This certificate must be in the current user's personal store.
`SigningTarget`	Required ITaskItem parameter. Specifies the files to sign with the certificate.
`TimestampUrl`	Optional `String` parameter. Specifies the URL of a time stamping server.
`TargetFrameworkVersion`	The version of the .NET Framework that is used for the target.

Remarks

In addition to the parameters listed above, this task inherits parameters from the Task class. For a list of these additional parameters and their descriptions, see Task base class.

Example

The following example uses the `SignFile` task to sign the files specified in the `FilesToSign` item collection with the certificate specified by the `Certificate` property.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <ItemGroup>
        <FileToSign Include="File.exe" />
    </ItemGroup>
    <PropertyGroup>
        <Certificate>Cert.cer</Certificate>
    </PropertyGroup>
    <Target Name="Sign">
        <SignFile
            CertificateThumbprint="$(CertificateThumbprint)"
            SigningTarget="@(FileToSign)"
            TargetFrameworkVersion="v4.5" />
    </Target>
</Project>
```

NOTE

The certificate thumbprint is the SHA-1 hash of the certificate. For more information, see Obtain the SHA-1 hash of a trusted root CA certificate. If you copy and paste the thumbprint from the certificate details, make sure you do not include the extra (3F) invisible character, which may prevent `SignFile` from finding the certificate.

See also

- Task reference
- Tasks

Touch task

2/21/2019 • 2 minutes to read • Edit Online

Sets the access and modification times of files.

Parameters

The following table describes the parameters of the `Touch` task.

PARAMETER	DESCRIPTION
`AlwaysCreate`	Optional `Boolean` parameter. If `true`, creates any files that do not already exist.
`Files`	Required ITaskItem `[]` parameter. Specifies the collection of files to touch.
`ForceTouch`	Optional `Boolean` parameter. If `true`, forces a file touch even if the files are read-only.
`Time`	Optional `String` parameter. Specifies a time other than the current time. The format must be a format that is acceptable to the Parse method.
`TouchedFiles`	Optional ITaskItem `[]` output parameter. Contains the collection of items that were successfully touched.

Remarks

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following example uses the `Touch` task to change the access and modification times of the files specified in the `Files` item collection, and puts the list of successfully touched files in the `FilesTouched` item collection.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

<ItemGroup>
    <Files Include="File1.cs;File2.cs;File3.cs" />
</ItemGroup>

    <Target Name="TouchFiles">
        <Touch
            Files="@(Files)">
            <Output
                TaskParameter="TouchedFiles"
                ItemName="FilesTouched"/>
    </Touch>
</Target>
</Project>
```

See also

- Tasks
- Task reference

UnregisterAssembly task

2/21/2019 • 2 minutes to read • Edit Online

Unregisters the specified assemblies for COM interop purposes. Performs the reverse of the RegisterAssembly task.

Parameters

The following table describes the parameters of the `UnregisterAssembly` task.

PARAMETER	DESCRIPTION
`Assemblies`	Optional ITaskItem `[]` parameter. Specifies the assemblies to be unregistered.
`AssemblyListFile`	Optional ITaskItem parameter. Contains information about the state between the `RegisterAssembly` task and the `UnregisterAssembly` task. This prevents the task from attempting to unregister an assembly that failed to register in the `RegisterAssembly` task. If this parameter is specified, the `Assemblies` and `TypeLibFiles` parameters are ignored.
`TypeLibFiles`	Optional ITaskItem `[]` output parameter. Unregisters the specified type library from the specified assembly. **Note:** This parameter is only necessary if the type library file name is different than the assembly name.

Remarks

It is not required that the assembly exists for this task to be successful. If you attempt to unregister an assembly that does not exist, the task will succeed with a warning. This occurs because it is the job of this task to remove the assembly registration from the registry. If the assembly does not exist, it is not in the registry, and therefore, the task succeeded.

In addition to the parameters listed above, this task inherits parameters from the AppDomainIsolatedTaskExtension class, which itself inherits from the MarshalByRefObject class. The `MarshalByRefObject` class provides the same functionality as the Task class, but it can be instantiated in its own application domain.

Example

The following example uses the `UnregisterAssembly` task to unregister the assembly at the path specified by the `OutputPath` and `FileName` properties, if it exists.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <PropertyGroup>
        <OutputPath>\Output\</OutputPath>
        <FileName>MyFile.dll</FileName>
    </PropertyGroup>
    <Target Name="UnregisterAssemblies">
        <UnregisterAssembly
            Condition="Exists('$(OutputPath)$(FileName)')"
            Assemblies="$(OutputPath)$(FileName)" />
    </Target>

</Project>
```

See also

- RegisterAssembly task
- Tasks
- Task reference

Unzip task

2/21/2019 • 2 minutes to read • Edit Online

Unzips a *.zip* archive to the specified location.

> **NOTE**
> The `Unzip` task is available in MSBuild 15.8 and above only.

Parameters

The following table describes the parameters of the `Unzip` task.

PARAMETER	DESCRIPTION
`DestinationFolder`	Required ITaskItem parameter Specifies the destination folder to unzip the file to.
`OverwriteReadOnlyFiles`	Optional `Boolean` parameter. If `true`, overwrites read-only files. Defaults to `false`.
`SkipUnchangedFiles`	Optional `Boolean` parameter. If `true`, skips unzipping files that are unchanged. Defaults to `true`. The `Unzip` task considers files to be unchanged if they have the same size and the same last modified time.
`SourceFiles`	Required ITaskItem `[]` parameter. Specifies one or more the files to unzip. When specifying multiple files they are unzipped in order to the same folder.

Remarks

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following example unzips an archive and overwrites any read-only files.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <Target Name="UnzipArchive" BeforeTargets="Build">
        <Unzip
            SourceFiles="MyArchive.zip"
            DestinationFolder="$(OutputPath)\unzipped"
            OverwriteReadOnlyFiles="true"
        />
    </Target>

</Project>
```

See also

- Tasks
- Task reference

UpdateManifest task

2/21/2019 • 2 minutes to read • Edit Online

Updates selected properties in a manifest and resigns.

Parameters

The following table describes the parameters of the `UpdateManifest` task.

PARAMETER	DESCRIPTION
`ApplicationManifest`	Required ITaskItem parameter. Specifies the application manifest.
`ApplicationPath`	Required `String` parameter. Specifies the path of the application manifest.
`InputManifest`	Required ITaskItem parameter. Specifies the manifest to update.
`OutputManifest`	Optional ITaskItem output parameter. Specifies the manifest that contains updated properties.

Remarks

In addition to having the parameters that are listed in the table, this task inherits parameters from the Task class. For a list of these additional parameters and their descriptions, see Task base class.

See also

- Tasks
- Task reference

Vbc task

4/16/2019 • 8 minutes to read • Edit Online

Wraps *vbc.exe*, which produces executables (*.exe*), dynamic-link libraries (*.dll*), or code modules (*.netmodule*). For more information on *vbc.exe*, see Visual Basic command-line compiler.

Parameters

The following table describes the parameters of the `Vbc` task.

PARAMETER	DESCRIPTION
`AdditionalLibPaths`	Optional `String[]` parameter. Specifies additional folders in which to look for assemblies specified in the References attribute.
`AddModules`	Optional `String[]` parameter. Causes the compiler to make all type information from the specified file(s) available to the project you are currently compiling. This parameter corresponds to the -addmodule switch of the *vbc.exe* compiler.
`BaseAddress`	Optional `String` parameter. Specifies the base address of the DLL. This parameter corresponds to the -baseaddress switch of the *vbc.exe* compiler.
`CodePage`	Optional `Int32` parameter. Specifies the code page to use for all source code files in the compilation. This parameter corresponds to the -codepage switch of the *vbc.exe* compiler.
`DebugType`	Optional `String[]` parameter. Causes the compiler to generate debugging information. This parameter can have the following values: - `full` - `pdbonly` The default value is `full`, which enables attaching a debugger to the running program. A value of `pdbonly` allows source code debugging when the program is started in the debugger, but displays assembly language code only when the running program is attached to the debugger. For more information, see -debug (Visual Basic).

PARAMETER	DESCRIPTION
`DefineConstants`	Optional `String[]` parameter. Defines conditional compiler constants. Symbol/value pairs are separated by semicolons and are specified with the following syntax: *symbol1* `=` *value1* `;` *symbol2* `=` *value2* This parameter corresponds to the -define switch of the *vbc.exe* compiler.
`DelaySign`	Optional `Boolean` parameter. If `true`, the task places the public key in the assembly. If `false`, the task fully signs the assembly. The default value is `false`.This parameter has no effect unless used with the `KeyFile` parameter or the `KeyContainer` parameter. This parameter corresponds to the -delaysign switch of the *vbc.exe* compiler.
`Deterministic`	Optional `Boolean` parameter. If `true`, causes the compiler to output an assembly whose binary content is identical across compilations if inputs are identical. For more information, see -deterministic.
`DisabledWarnings`	Optional `String` parameter. Suppresses the specified warnings. You only need to specify the numeric part of the warning identifier. Multiple warnings are separated by semicolons. This parameter corresponds to the -nowarn switch of the *vbc.exe* compiler.
`DocumentationFile`	Optional `String` parameter. Processes documentation comments to the specified XML file. This parameter overrides the `GenerateDocumentation` attribute. For more information, see -doc.
`EmitDebugInformation`	Optional `Boolean` parameter. If `true`, the task generates debugging information and places it in a *.pdb* file. For more information, see -debug (Visual Basic).

PARAMETER	DESCRIPTION
`ErrorReport`	Optional `String` parameter. Specifies how the task should report internal compiler errors. This parameter can have the following values: - `prompt` - `send` - `none` If `prompt` is specified and an internal compiler error occurs, the user is prompted with an option of whether to send the error data to Microsoft. If `send` is specified and an internal compiler error occurs, the task sends the error data to Microsoft. The default value is `none`, which reports errors in text output only. This parameter corresponds to the -errorreport switch of the *vbc.exe* compiler.
`FileAlignment`	Optional `Int32` parameter. Specifies, in bytes, where to align the sections of the output file. This parameter can have the following values: - `512` - `1024` - `2048` - `4096` - `8192` This parameter corresponds to the -filealign switch of the *vbc.exe* compiler.
`GenerateDocumentation`	Optional `Boolean` parameter. If `true`, generates documentation information and places it in an XML file with the name of the executable file or library that the task is creating. For more information, see -doc.
`Imports`	Optional ITaskItem`[]` parameter. Imports namespaces from the specified item collections. This parameter corresponds to the -imports switch of the *vbc.exe* compiler.
`KeyContainer`	Optional `String` parameter. Specifies the name of the cryptographic key container. This parameter corresponds to the -keycontainer switch of the *vbc.exe* compiler.

PARAMETER	DESCRIPTION
`KeyFile`	Optional `String` parameter. Specifies the file name containing the cryptographic key. For more information, see -keyfile.
`LangVersion`	Optional System.String parameter. Specifies the language version, such as "15.5".
`LinkResources`	Optional ITaskItem `[]` parameter. Creates a link to a .NET Framework resource in the output file; the resource file is not placed in the output file. This parameter corresponds to the -linkresource switch of the *vbc.exe* compiler.
`MainEntryPoint`	Optional `String` parameter. Specifies the class or module that contains the `Sub Main` procedure. This parameter corresponds to the -main switch of the *vbc.exe* compiler.
`ModuleAssemblyName`	Optional `String` parameter. Specifies the assembly that this module is a part of.
`NoConfig`	Optional `Boolean` parameter. Specifies that the compiler should not use the *vbc.rsp* file. This parameter corresponds to the -noconfig parameter of the *vbc.exe* compiler.
`NoLogo`	Optional `Boolean` parameter. If `true`, suppresses display of compiler banner information. This parameter corresponds to the -nologo switch of the *vbc.exe* compiler.
`NoStandardLib`	Optional `Boolean` parameter. Causes the compiler not to reference the standard libraries. This parameter corresponds to the -nostdlib switch of the *vbc.exe* compiler.
`NoVBRuntimeReference`	Optional `Boolean` parameter. Internal use only. If true, prevents the automatic reference to *Microsoft.VisualBasic.dll*.
`NoWarnings`	Optional `Boolean` parameter. If `true`, the task suppresses all warnings. For more information, see -nowarn.

PARAMETER	DESCRIPTION
`Optimize`	Optional `Boolean` parameter. If `true`, enables compiler optimizations. This parameter corresponds to the -optimize switch of the *vbc.exe* compiler.
`OptionCompare`	Optional `String` parameter. Specifies how string comparisons are made. This parameter can have the following values: - `binary` - `text` The value `binary` specifies that the task uses binary string comparisons. The value `text` specifies that the task uses text string comparisons. The default value of this parameter is `binary`. This parameter corresponds to the -optioncompare switch of the *vbc.exe* compiler.
`OptionExplicit`	Optional `Boolean` parameter. If `true`, explicit declaration of variables is required. This parameter corresponds to the -optionexplicit switch of the *vbc.exe* compiler.
`OptionInfer`	Optional `Boolean` parameter. If `true`, allows type inference of variables.
`OptionStrict`	Optional `Boolean` parameter. If `true`, the task enforces strict type semantics to restrict implicit type conversions. This parameter corresponds to the -optionstrict switch of the *vbc.exe* compiler.
`OptionStrictType`	Optional `String` parameter. Specifies which strict type semantics generate a warning. Currently, only "custom" is supported. This parameter corresponds to the -optionstrict switch of the *vbc.exe* compiler.
`OutputAssembly`	Optional `String` output parameter. Specifies the name of the output file. This parameter corresponds to the -out switch of the *vbc.exe* compiler.
`Platform`	Optional `String` parameter. Specifies the processor platform to be targeted by the output file. This parameter can have a value of `x86`, `x64`, `Itanium`, or `anycpu`. Default is `anycpu`. This parameter corresponds to the -platform switch of the *vbc.exe* compiler.

PARAMETER	DESCRIPTION
`References`	Optional ITaskItem `[]` parameter. Causes the task to import public type information from the specified items into the current project. This parameter corresponds to the -reference switch of the *vbc.exe* compiler.
`RemoveIntegerChecks`	Optional `Boolean` parameter. If `true`, disables integer overflow error checks. The default value is `false`. This parameter corresponds to the -removeintchecks switch of the *vbc.exe* compiler.
`Resources`	Optional ITaskItem `[]` parameter. Embeds a .NET Framework resource into the output file. This parameter corresponds to the -resource switch of the *vbc.exe* compiler.
`ResponseFiles`	Optional ITaskItem `[]` parameter. Specifies the response file that contains commands for this task. This parameter corresponds to the @ (Specify Response File) option of the *vbc.exe* compiler.
`RootNamespace`	Optional `String` parameter. Specifies the root namespace for all type declarations. This parameter corresponds to the -rootnamespace switch of the *vbc.exe* compiler.
`SdkPath`	Optional `String` parameter. Specifies the location of *mscorlib.dll* and *microsoft.visualbasic.dll*. This parameter corresponds to the -sdkpath switch of the *vbc.exe* compiler.
`Sources`	Optional ITaskItem `[]` parameter. Specifies one or more Visual Basic source files.
`TargetCompactFramework`	Optional `Boolean` parameter. If `true`, the task targets the .NET Compact Framework. This switch corresponds to the -netcf switch of the *vbc.exe* compiler.
`TargetType`	Optional `String` parameter. Specifies the file format of the output file. This parameter can have a value of `library`, which creates a code library, `exe`, which creates a console application, `module`, which creates a module, or `winexe`, which creates a Windows program. Default is `library`. This parameter corresponds to the -target switch of the *vbc.exe* compiler.

PARAMETER	DESCRIPTION
`Timeout`	Optional `Int32` parameter. Specifies the amount of time, in milliseconds, after which the task executable is terminated. The default value is `Int.MaxValue`, indicating that there is no time out period.
`ToolPath`	Optional `String` parameter. Specifies the location from where the task will load the underlying executable file (*vbc.exe*). If this parameter is not specified, the task uses the SDK installation path corresponding to the version of the framework that is running MSBuild.
`TreatWarningsAsErrors`	Optional `Boolean` parameter. If `true`, all warnings are treated as errors. For more information, see -warnaserror (Visual Basic).
`UseHostCompilerIfAvailable`	Optional `Boolean` parameter. Instructs the task to use the in-process compiler object, if available. Used only by Visual Studio.
`Utf8Output`	Optional `Boolean` parameter. Logs compiler output using UTF-8 encoding. This parameter corresponds to the -utf8output switch of the *vbc.exe* compiler.
`Verbosity`	Optional `String` parameter. Specifies the verbosity of the compiler's output. Verbosity can be `Quiet`, `Normal` (the default), or `Verbose`.
`WarningsAsErrors`	Optional `String` parameter. Specifies a list of warnings to treat as errors. For more information, see -warnaserror (Visual Basic). This parameter overrides the `TreatWarningsAsErrors` parameter.
`WarningsNotAsErrors`	Optional `String` parameter. Specifies a list of warnings that are not treated as errors. For more information, see -warnaserror (Visual Basic). This parameter is only useful if the `TreatWarningsAsErrors` parameter is set to `true`.
`Win32Icon`	Optional `String` parameter. Inserts an *.ico* file in the assembly, which gives the output file the desired appearance in **File Explorer**. This parameter corresponds to the -win32icon switch of the *vbc.exe* compiler.

PARAMETER	DESCRIPTION
`Win32Resources`	Optional `String` parameter. Inserts a Win32 resource (*.res*) file in the output file. This parameter corresponds to the -win32resource switch of the *vbc.exe* compiler.

Remarks

In addition to the parameters listed above, this task inherits parameters from the ToolTaskExtension class, which itself inherits from the ToolTask class. For a list of these additional parameters and their descriptions, see ToolTaskExtension base class.

Example

The following example compiles a Visual Basic project.

```
<VBC
   Sources="@(sources)"
   Resources="strings.resources"
   Optimize="true"
   OutputAssembly="out.exe"/>
```

See also

- Visual Basic command-line compiler
- Tasks
- Task reference

VerifyFileHash task

9/27/2019 • 2 minutes to read • Edit Online

Verifies that a file matches the expected file hash.

This task was added in 15.8, but requires a workaround to use for MSBuild versions below 16.0.

Task parameters

The following table describes the parameters of the `VerifyFileHash` task.

PARAMETER	DESCRIPTION
`File`	Required `String` parameter. The file to be hashed and validated.
`Hash`	Required `String` parameter. The expected hash of the file.
`Algorithm`	Optional `String` parameter. The algorithm. Allowed values: `SHA256`, `SHA384`, `SHA512`. Default = `SHA256`.
`HashEncoding`	Optional `String` parameter. The encoding to use for generated hashes. Defaults to `hex`. Allowed values = `hex`, `base64`.

Example

The following example uses the `VerifyFileHash` task to verify its own checksum.

```
<Project>
  <Target Name="VerifyHash">
    <GetFileHash Files="$(MSBuildProjectFullPath)">
      <Output
          TaskParameter="Items"
          ItemName="FilesWithHashes" />
    </GetFileHash>

    <Message Importance="High"
             Text="@(FilesWithHashes->'%(Identity): %(FileHash)')" />

    <VerifyFileHash File="$(MSBuildThisFileFullPath)"
                    Hash="$(ExpectedHash)" />
  </Target>
</Project>
```

See also

- Tasks
- Task reference

Warning task

4/16/2019 • 2 minutes to read • Edit Online

Logs a warning during a build based on an evaluated conditional statement.

Parameters

The following table describes the parameters of the `Warning` task.

PARAMETER	DESCRIPTION
`Code`	Optional `String` parameter. The warning code to associate with the warning.
`File`	Optional `String` parameter. Specifies the relevant file, if any. If no file is provided, the file containing the Warning task is used.
`HelpKeyword`	Optional `String` parameter. The Help keyword to associate with the warning.
`Text`	Optional `String` parameter. The warning text that MSBuild logs if the `Condition` parameter evaluates to `true`.

Remarks

The `Warning` task allows MSBuild projects to check for the presence of a required configuration or property before proceeding with the next build step.

If the `Condition` parameter of the `Warning` task evaluates to `true`, the value of the `Text` parameter is logged and the build continues to execute. If a `Condition` parameter does not exist, the warning text is logged. For more information on logging, see Obtain build logs.

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following code example checks for properties that are set on the command line. If there are no properties set, the project raises a warning event, and logs the value of the `Text` parameter of the `Warning` task.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
    <Target Name="ValidateCommandLine">
        <Warning
            Text=" The 0 property was not set on the command line."
            Condition="'$(0)' == ''" />
        <Warning
            Text=" The FREEBUILD property was not set on the command line."
            Condition="'$(FREEBUILD)' == ''" />
    </Target>
    ...
</Project>
```

See also

- Obtain build logs
- Project file schema reference

WriteCodeFragment task

2/21/2019 • 2 minutes to read • Edit Online

Generates a temporary code file from the specified generated code fragment. Does not delete the file.

Parameters

The following table describes the parameters of the `WriteCodeFragment` task.

PARAMETER	DESCRIPTION
`AssemblyAttributes`	Optional ITaskItem `[]` parameter. Description of the attributes to write. The item `Include` value is the full type name of the attribute, for example, "System.AssemblyVersionAttribute". Each metadata is the name-value pair of a parameter, which must be of type `String`. Some attributes only allow positional constructor arguments. However, you can use such arguments in any attribute. To set positional constructor attributes, use metadata names that resemble "_Parameter1", "_Parameter2", and so on. A parameter index cannot be skipped.
`Language`	Required `String` parameter. Specifies the language of the code to generate. `Language` can be any language for which a CodeDom provider is available, for example, "C#" or "VisualBasic". The emitted file will have the default file name extension for that language.
`OutputDirectory`	Optional ITaskItem parameter. Specifies the destination folder for the generated code, typically the intermediate folder.
`OutputFile`	Optional ITaskItem output parameter. Specifies the path of the file that was generated. If this parameter is set by using a file name, the destination folder is prepended to the file name. If it is set by using a root, the destination folder is ignored. If this parameter is not set, the output file name is the destination folder, an arbitrary file name, and the default file name extension for the specified language.

Remarks

In addition to having the parameters that are listed in the table, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their

descriptions, see TaskExtension base class.

See also

- Tasks
- Task reference

WriteLinesToFile task

2/21/2019 • 2 minutes to read • Edit Online

Writes the paths of the specified items to the specified text file.

Task parameters

The following table describes the parameters of the `WriteLinestoFile` task.

PARAMETER	DESCRIPTION
`File`	Required ITaskItem parameter. Specifies the file to write the items to.
`Lines`	Optional ITaskItem `[]` parameter. Specifies the items to write to the file.
`Overwrite`	Optional `Boolean` parameter. If `true`, the task overwrites any existing content in the file.
`Encoding`	Optional `String` parameter. Selects the character encoding, for example, "Unicode". See also Encoding.
`WriteOnlyWhenDifferent`	Optional `Boolean` parameter. If `true`, the target file specified, if it exists, will be read first to compare against what the task would have written. If identical, the file is not written to disk and the timestamp will be preserved.

Remarks

If `Overwrite` is `true`, creates a new file, write the contents to the file, and then closes the file. If the target file already exists, it is overwritten. If `Overwrite` is `false`, appends the contents to file, creating the target file if it does not already exist.

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following example uses the `WriteLinesToFile` task to write the paths of the items in the `MyItems` item collection to the file specified by the `MyTextFile` item collection.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <ItemGroup>
        <MyTextFile Include="Items.txt"/>
        <MyItems Include="*.cs"/>
    </ItemGroup>

    <Target Name="WriteToFile">
        <WriteLinesToFile
            File="@(MyTextFile)"
            Lines="@(MyItems)"
            Overwrite="true"
            Encoding="Unicode"/>
    </Target>

</Project>
```

In this example we use a property with embedded newlines to write a text file with multiple lines. If an entry in `Lines` has embedded newline characters, the new lines will be included in the output file. In this way, you can reference multi-line properties.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>netcoreapp2.1</TargetFramework>
  </PropertyGroup>

  <Target Name="WriteLaunchers" AfterTargets="CopyFilesToOutputDirectory">
      <PropertyGroup>
        <LauncherCmd>
@ECHO OFF
dotnet %~dp0$(AssemblyName).dll %*
        </LauncherCmd>
      </PropertyGroup>

      <WriteLinesToFile
        File="$(OutputPath)$(AssemblyName).cmd"
        Overwrite="true"
        Lines="$(LauncherCmd)" />
  </Target>
</Project>
```

See also

- Tasks
- Task reference

XmlPeek task

2/21/2019 • 2 minutes to read • Edit Online

Returns values as specified by XPath Query from an XML file.

Parameters

The following table describes the parameters of the `XmlPeek` task.

PARAMETER	DESCRIPTION
`Namespaces`	Optional `String` parameter. Specifies the namespaces for the XPath query prefixes.
`Query`	Optional `String` parameter. Specifies the XPath query.
`Result`	Optional ITaskItem `[]` output parameter. Contains the results that are returned by this task.
`XmlContent`	Optional `String` parameter. Specifies the XML input as a string.
`XmlInputPath`	Optional ITaskItem parameter. Specifies the XML input as a file path.

Remarks

In addition to having the parameters that are listed in the table, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

See also

- Tasks
- Task reference

XmlPoke task

2/21/2019 • 2 minutes to read • Edit Online

Sets values as specified by an XPath query into an XML file.

Parameters

The following table describes the parameters of the `XmlPoke` task.

PARAMETER	DESCRIPTION
`Namespaces`	Optional `String` parameter. Specifies the namespaces for XPath query prefixes. `Namespaces` is an XML snippet consisting of `Namespace` elements with attributes `Prefix` and `Uri`. The attribute `Prefix` specifies the prefix to associate with the namespace specified in `Uri` attribute. Do not use an empty `Prefix`.
`Query`	Optional `String` parameter. Specifies the XPath query.
`Value`	Required ITaskItem parameter. Specifies the value to be inserted into the specified path.
`XmlInputPath`	Optional ITaskItem parameter. Specifies the XML input as a file path.

Remarks

In addition to having the parameters that are listed in the table, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

Here is a sample.xml to modify:

```
<Package xmlns="http://schemas.microsoft.com/appx/manifest/foundation/windows10"
         xmlns:mp="http://schemas.microsoft.com/appx/2014/phone/manifest"
         xmlns:uap="http://schemas.microsoft.com/appx/manifest/uap/windows10" >
<Identity Name="Sample.Product " Publisher="CN=1234" Version="1.0.0.0" />
<mp:PhoneIdentity PhoneProductId="456" PhonePublisherId="0" />
</Package>
```

In this example, if you want to modify `/Package/mp:PhoneIdentity/PhonePublisherId`, then use

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <Namespace>
        <Namespace Prefix="dn" Uri="http://schemas.microsoft.com/appx/manifest/foundation/windows10" />
        <Namespace Prefix="mp" Uri="http://schemas.microsoft.com/appx/2014/phone/manifest" />
        <Namespace Prefix="uap" Uri="http://schemas.microsoft.com/appx/manifest/uap/windows10" />
    </Namespace>
</PropertyGroup>

<Target Name="Poke">
  <XmlPoke
    XmlInputPath="Sample.xml"
    Value="MyId"
    Query="/dn:Package/mp:PhoneIdentity/@PhoneProductId"
    Namespaces="$(Namespace)"/>
</Target>
</Project>
```

`dn` is here used as an artificial namespace prefix for default namespace.

See also

- Tasks
- Task reference

XslTransformation task

2/21/2019 • 2 minutes to read • Edit Online

Transforms an XML input by using an XSLT or compiled XSLT and outputs to an output device or a file.

Parameters

The following table describes the parameters of the `XslTransformation` task.

PARAMETER	DESCRIPTION
`OutputPaths`	Required ITaskItem `[]` parameter. Specifies the output files for the XML transformation.
`Parameters`	Optional `String` parameter. Specifies the parameters to the XSLT Input document.
`XmlContent`	Optional `String` parameter. Specifies the XML input as a string.
`XmlInputPaths`	Optional ITaskItem `[]` parameter. Specifies the XML input files.
`XslCompiledDllPath`	Optional ITaskItem parameter. Specifies the compiled XSLT.
`XslContent`	Optional `String` parameter. Specifies the XSLT input as a string.
`XslInputPath`	Optional ITaskItem parameter. Specifies the XSLT input file.

Remarks

In addition to having the parameters that are listed in the table, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

See also

- Tasks
- Task reference

ZipDirectory task

2/21/2019 • 2 minutes to read • Edit Online

Creates a *.zip* archive from the contents of a directory.

> **NOTE**
> The `ZipDirectory` task is available in MSBuild 15.8 and above only.

Parameters

The following table describes the parameters of the `ZipDirectory` task.

PARAMETER	DESCRIPTION
`DestinationFile`	Required ITaskItem parameter The full path to the *.zip* file to create.
`Overwrite`	Optional `Boolean` parameter. If `true`, skips the destination file will be overwritten if it exists. Defaults to `false`.
`SourceDirectory`	Required ITaskItem parameter. Specifies the directory to create a *.zip* archive from.

Remarks

In addition to the parameters listed above, this task inherits parameters from the TaskExtension class, which itself inherits from the Task class. For a list of these additional parameters and their descriptions, see TaskExtension base class.

Example

The following example creates a *.zip* archive from the output directory after building a project.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

    <Target Name="ZipOutputPath" AfterTargets="Build">
        <ZipDirectory
            SourceDirectory="$(OutputPath)"
            DestinationFile="$(MSBuildProjectDirectory)\output.zip" />
    </Target>

</Project>
```

See also

- Tasks

- Task reference

MSBuild conditions

10/24/2019 • 2 minutes to read • Edit Online

MSBuild supports a specific set of conditions that can be applied wherever a `Condition` attribute is allowed. The following table explains those conditions.

CONDITION	DESCRIPTION
'`stringA`' == '`stringB`'	Evaluates to `true` if `stringA` equals `stringB`. For example: `Condition="'$(CONFIG)'=='DEBUG'"` Single quotes are not required for simple alphanumeric strings or boolean values. However, single quotes are required for empty values.
'`stringA`' != '`stringB`'	Evaluates to `true` if `stringA` is not equal to `stringB`. For example: `Condition="'$(CONFIG)'!='DEBUG'"` Single quotes are not required for simple alphanumeric strings or boolean values. However, single quotes are required for empty values.
<, >, <=, >=	Evaluates the numeric values of the operands. Returns `true` if the relational evaluation is true. Operands must evaluate to a decimal or hexadecimal number. Hexadecimal numbers must begin with "0x". **Note:** In XML, the characters `<` and `>` must be escaped. The symbol `<` is represented as `<`. The symbol `>` is represented as `>`.
Exists('`stringA`')	Evaluates to `true` if a file or folder with the name `stringA` exists. For example: `Condition="!Exists('$(builtdir)')"` Single quotes are not required for simple alphanumeric strings or boolean values. However, single quotes are required for empty values.

CONDITION	DESCRIPTION
HasTrailingSlash('`stringA`')	Evaluates to `true` if the specified string contains either a trailing backward slash (\) or forward slash (/) character. For example: `Condition="!HasTrailingSlash('$(OutputPath)')"` Single quotes are not required for simple alphanumeric strings or boolean values. However, single quotes are required for empty values.
!	Evaluates to `true` if the operand evaluates to `false`.
And	Evaluates to `true` if both operands evaluate to `true`.
Or	Evaluates to `true` if at least one of the operands evaluates to `true`.
()	Grouping mechanism that evaluates to `true` if expressions contained inside evaluate to `true`.
if (%expression%), $else$, $endif$	Checks whether the specified `%expression%` matches the string value of the passed custom template parameter. If the `$if$` condition evaluates to `true`, then its statements are run; otherwise, the `$else$` condition is checked. If the `$else$` condition is `true`, then its statements are run; otherwise, the `$endif$` condition ends expression evaluation. For examples of usage, see Visual Studio project/item template parameter logic.

See also

- MSBuild reference
- Conditional constructs
- Walkthrough: Creating an MSBuild project file from scratch

MSBuild conditional constructs

2/21/2019 • 2 minutes to read • Edit Online

MSBuild provides a mechanism for either/or processing with the Choose, When, and Otherwise elements.

Use the Choose element

The `Choose` element contains a series of `When` elements with `Condition` attributes that are tested in order from top to bottom until one evaluates to `true`. If more than one `When` element evaluates to `true`, only the first one is used. An `Otherwise` element, if present, will be evaluated if no condition on a `When` element evaluates to `true` .

`Choose` elements can be used as child elements of `Project`, `When` and `Otherwise` elements. `When` and `Otherwise` elements can have `ItemGroup`, `PropertyGroup`, or `Choose` child elements.

Example

The following example uses the `Choose` and `When` elements for either/or processing. The properties and items for the project are set depending on the value of the `Configuration` property.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003" >
    <PropertyGroup>
        <Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>
        <OutputType>Exe</OutputType>
        <RootNamespace>ConsoleApplication1</RootNamespace>
        <AssemblyName>ConsoleApplication1</AssemblyName>
        <WarningLevel>4</WarningLevel>
    </PropertyGroup>
    <Choose>
        <When Condition=" '$(Configuration)'=='Debug' ">
            <PropertyGroup>
                <DebugSymbols>true</DebugSymbols>
                <DebugType>full</DebugType>
                <Optimize>false</Optimize>
                <OutputPath>.\bin\Debug\</OutputPath>
                <DefineConstants>DEBUG;TRACE</DefineConstants>
            </PropertyGroup>
            <ItemGroup>
                <Compile Include="UnitTesting\*.cs" />
                <Reference Include="NUnit.dll" />
            </ItemGroup>
        </When>
        <When Condition=" '$(Configuration)'=='retail' ">
            <PropertyGroup>
                <DebugSymbols>false</DebugSymbols>
                <Optimize>true</Optimize>
                <OutputPath>.\bin\Release\</OutputPath>
                <DefineConstants>TRACE</DefineConstants>
            </PropertyGroup>
        </When>
    </Choose>
    <!-- Rest of Project -->
</Project>
```

See also

- Choose element (MSBuild)
- When element (MSBuild)
- Otherwise element (MSBuild)
- MSBuild reference

MSBuild reserved and well-known properties

11/26/2019 • 5 minutes to read • Edit Online

MSBuild provides a set of predefined properties that store information about the project file and the MSBuild binaries. These properties are evaluated in the same manner as other MSBuild properties. For example, to use the `MSBuildProjectFile` property, you type `$(MSBuildProjectFile)`.

MSBuild uses the values in the following table to predefine reserved and well-known properties. Reserved properties cannot be overridden, but well-known properties can be overridden by using identically named environment properties, global properties, or properties that are declared in the project file.

Reserved and well-known properties

The following table describes the MSBuild predefined properties.

PROPERTY	RESERVED OR WELL-KNOWN	DESCRIPTION
`MSBuildBinPath`	Reserved	The absolute path of the folder where the MSBuild binaries that are currently being used are located (for example, *C:\Windows\Microsoft.Net\Framework\<versionNumber>*). This property is useful if you have to refer to files in the MSBuild directory. Do not include the final backslash on this property.
`MSBuildExtensionsPath`	Well-known	Introduced in the .NET Framework 4: there is no difference between the default values of `MSBuildExtensionsPath` and `MSBuildExtensionsPath32`. You can set the environment variable `MSBUILDLEGACYEXTENSIONSPATH` to a non-null value to enable the behavior of the default value of `MSBuildExtensionsPath` in earlier versions. In the .NET Framework 3.5 and earlier, the default value of `MSBuildExtensionsPath` points to the path of the MSBuild subfolder under the *\Program Files* or *\Program Files (x86)* folder, depending on the bitness of the current process. For example, for a 32-bit process on a 64-bit machine, this property points to the *\Program Files (x86)* folder. For a 64-bit process on a 64-bit machine, this property points to the *\Program Files* folder. Do not include the final backslash on this property. This location is a useful place to put custom target files. For example, your target files could be installed at *\Program Files\MSBuild\MyFiles\Northwind.targets* and then imported in project files by using this XML code: `<Import Project="$(MSBuildExtensionsPath)\MyFiles\Northwi`

PROPERTY	RESERVED OR WELL-KNOWN	DESCRIPTION
`MSBuildExtensionsPath32`	Well-known	The path of the MSBuild subfolder under the *\Program Files* or *\Program Files (x86)* folder. The path always points to the 32-bit *\Program Files (x86)* folder on a 32-bit machine and *\Program Files* on a 64-bit machine.". See also `MSBuildExtensionsPath` and `MSBuildExtensionsPath64`. Do not include the final backslash on this property.
`MSBuildExtensionsPath64`	Well-known	The path of the MSBuild subfolder under the *\Program Files* folder. For a 64-bit machine, this path always points to the *\Program Files* folder. For a 32-bit machine, this path is blank. See also `MSBuildExtensionsPath` and `MSBuildExtensionsPath32`. Do not include the final backslash on this property.
`MSBuildLastTaskResult`	Reserved	`true` if the previous task completed without any errors (even if there were warnings), or `false` if the previous task had errors. Typically, when an error occurs in a task, the error is the last thing that happens in that project. Therefore, the value of this property is never `false`, except in these scenarios: - When the `ContinueOnError` attribute of the Task element (MSBuild) is set to `WarnAndContinue` (or `true`) or `ErrorAndContinue`. - When the `Target` has an OnError element (MSBuild) as a child element.
`MSBuildNodeCount`	Reserved	The maximum number of concurrent processes that are used when building. This is the value that you specified for **-maxcpucount** on the command line. If you specified **-maxcpucount** without specifying a value, then `MSBuildNodeCount` specifies the number of processors in the computer. For more information, see Command-line reference and Build multiple projects in parallel.
`MSBuildProgramFiles32`	Reserved	The location of the 32-bit program folder; for example, *C:\Program Files (x86)*. Do not include the final backslash on this property.
`MSBuildProjectDefaultTargets`	Reserved	The complete list of targets that are specified in the `DefaultTargets` attribute of the `Project` element. For example, the following `Project` element would have an `MSBuildDefaultTargets` property value of `A;B;C`: `<Project DefaultTargets="A;B;C" >`
`MSBuildProjectDirectory`	Reserved	The absolute path of the directory where the project file is located, for example *C:\MyCompany\MyProduct*. Do not include the final backslash on this property.

PROPERTY	RESERVED OR WELL-KNOWN	DESCRIPTION
`MSBuildProjectDirectoryNoRoot`	Reserved	The value of the `MSBuildProjectDirectory` property, excluding the root drive. Do not include the final backslash on this property.
`MSBuildProjectExtension`	Reserved	The file name extension of the project file, including the period; for example, *.proj*.
`MSBuildProjectFile`	Reserved	The complete file name of the project file, including the file name extension; for example, *MyApp.proj*.
`MSBuildProjectFullPath`	Reserved	The absolute path and complete file name of the project file, including the file name extension; for example, *C:\MyCompany\MyProduct\MyApp.proj*.
`MSBuildProjectName`	Reserved	The file name of the project file without the file name extension; for example, *MyApp*.
`MSBuildRuntimeType`	Reserved	The type of the runtime that is currently executing. Introduced in MSBuild 15. Value may be undefined (prior to MSBuild 15), `Full` indicating that MSBuild is running on the desktop .NET Framework, `Core` indicating that MSBuild is running on .NET Core (for example in `dotnet build`), or `Mono` indicating that MSBuild is running on Mono.

`MSBuildStartupDirectory`	Reserved	The absolute path of the folder where MSBuild is called. By using this property, you can build everything below a specific point in a project tree without creating *<dirs>.proj* files in every directory. Instead, you have just one project—for example, *c:\traversal.proj*, as shown here:
		
		`<Project ...> <ItemGroup> <ProjectFiles Include="$ (MSBuildStartupDirectory) **\*.csproj"/> </ItemGroup> <Target Name="build"> <MSBuild Projects="@(ProjectFiles)"/> </Target> </Project>`
		
		To build at any point in the tree, type:
		
		`msbuild c:\traversal.proj`
		
		Do not include the final backslash on this property.

`MSBuildThisFile`	Reserved	The file name and file extension portion of `MSBuildThisFileFullPath`.
`MSBuildThisFileDirectory`	Reserved	The directory portion of `MSBuildThisFileFullPath`. Include the final backslash in the path.
`MSBuildThisFileDirectoryNoRoot`	Reserved	The directory portion of `MSBuildThisFileFullPath`, excluding the root drive. Include the final backslash in the path.
`MSBuildThisFileExtension`	Reserved	The file name extension portion of `MSBuildThisFileFullPath`.

PROPERTY	RESERVED OR WELL-KNOWN	DESCRIPTION
`MSBuildThisFileFullPath`	Reserved	The absolute path of the project or targets file that contains the target that is running. Tip: You can specify a relative path in a targets file that's relative to the targets file and not relative to the original project file.
`MSBuildThisFileName`	Reserved	The file name portion of `MSBuildThisFileFullPath`, without the file name extension.
`MSBuildToolsPath`	Reserved	The installation path of the MSBuild version that's associated with the value of `MSBuildToolsVersion`. Do not include the final backslash in the path. This property cannot be overridden.
`MSBuildToolsVersion`	Reserved	The version of the MSBuild Toolset that is used to build the project. Note: An MSBuild Toolset consists of tasks, targets, and tools that are used to build an application. The tools include compilers such as *csc.exe* and *vbc.exe*. For more information, see Toolset (ToolsVersion), and Standard and custom Toolset configurations.
`MSBuildVersion`	Reserved	The version of MSBuild used to build the project. This property can't be overridden, otherwise the error message `MSB4004 - The 'MSBuildVersion' property is reserved, and can not be modified.` is returned.

Names that conflict with MSBuild elements

In addition to the above, names corresponding to MSBuild language elements cannot be used for user-defined properties, items, or item metadata:

- VisualStudioProject
- Target
- PropertyGroup
- Output
- ItemGroup
- UsingTask
- ProjectExtensions
- OnError
- ImportGroup
- Choose
- When
- Otherwise

See also

- MSBuild reference
- MSBuild properties

Common MSBuild project properties

10/21/2019 • 13 minutes to read • Edit Online

The following table lists frequently used properties that are defined in the Visual Studio project files or included in *.targets* files that MSBuild provides.

Project files in Visual Studio (*.csproj*, *.vbproj*, *.vcxproj*, and others) contain MSBuild XML code that runs when you build a project by using the IDE. Projects typically import one or more *.targets* files to define their build process. For more information, see MSBuild .targets files.

List of common properties and parameters

PROPERTY OR PARAMETER NAME	DESCRIPTION
AdditionalLibPaths	Specifies additional folders in which compilers should look for reference assemblies.
AddModules	Causes the compiler to make all type information from the specified files available to the project you are compiling. This property is equivalent to the `/addModules` compiler switch.
ALToolPath	The path where *AL.exe* can be found. This property overrides the current version of *AL.exe* to enable use of a different version.
ApplicationIcon	The *.ico* icon file to pass to the compiler for embedding as a Win32 icon. The property is equivalent to the `/win32icon` compiler switch.
ApplicationManifest	Specifies the path of the file that is used to generate external User Account Control (UAC) manifest information. Applies only to Visual Studio projects targeting Windows Vista. In most cases, the manifest is embedded. However, if you use Registration Free COM or ClickOnce deployment, then the manifest can be an external file that is installed together with your application assemblies. For more information, see the NoWin32Manifest property in this topic.
AssemblyOriginatorKeyFile	Specifies the file that's used to sign the assembly (*.snk* or *.pfx*) and that's passed to the ResolveKeySource task to generate the actual key that's used to sign the assembly.
AssemblySearchPaths	A list of locations to search during build-time reference assembly resolution. The order in which paths appear in this list is meaningful because paths listed earlier takes precedence over later entries.
AssemblyName	The name of the final output assembly after the project is built.
BaseAddress	Specifies the base address of the main output assembly. This property is equivalent to the `/baseaddress` compiler switch.

PROPERTY OR PARAMETER NAME	DESCRIPTION
BaseOutputPath	Specifies the base path for the output file. If it is set, MSBuild will use `OutputPath = $(BaseOutputPath)\$(Configuration)\`. Example syntax: `<BaseOutputPath>c:\xyz\bin\</BaseOutputPath>`
BaseIntermediateOutputPath	The top-level folder where all configuration-specific intermediate output folders are created. The default value is `obj\`. The following code is an example: `<BaseIntermediateOutputPath>c:\xyz\obj\ </BaseIntermediateOutputPath>`
BuildInParallel	A boolean value that indicates whether project references are built or cleaned in parallel when Multi-Proc MSBuild is used. The default value is `true`, which means that projects will be built in parallel if the system has multiple cores or processors.
BuildProjectReferences	A boolean value that indicates whether project references are built by MSBuild. Automatically set to `false` if you are building your project in the Visual Studio integrated development environment (IDE), `true` if otherwise. `-p:BuildProjectReferences=false` can be specified on the command line to avoid checking that referenced projects are up to date.
CleanFile	The name of the file that will be used as the "clean cache." The clean cache is a list of generated files to be deleted during the cleaning operation. The file is put in the intermediate output path by the build process. This property specifies only file names that do not have path information.
CodePage	Specifies the code page to use for all source-code files in the compilation. This property is equivalent to the `/codepage` compiler switch.
CompilerResponseFile	An optional response file that can be passed to the compiler tasks.
Configuration	The configuration that you are building, either "Debug" or "Release."
CscToolPath	The path of *csc.exe*, the Visual C# compiler.
CustomBeforeMicrosoftCommonTargets	The name of a project file or targets file that is to be imported automatically before the common targets import.
DebugSymbols	A boolean value that indicates whether symbols are generated by the build. Setting **-p:DebugSymbols=false** on the command line disables generation of program database (*.pdb*) symbol files.

PROPERTY OR PARAMETER NAME	DESCRIPTION
DebugType	Defines the level of debug information that you want generated. Valid values are "full," "pdbonly," "portable", "embedded", and "none."
DefineConstants	Defines conditional compiler constants. Symbol/value pairs are separated by semicolons and are specified by using the following syntax: *symbol1 = value1 ; symbol2 = value2* The property is equivalent to the `/define` compiler switch.
DefineDebug	A boolean value that indicates whether you want the DEBUG constant defined.
DefineTrace	A boolean value that indicates whether you want the TRACE constant defined.
DelaySign	A boolean value that indicates whether you want to delay-sign the assembly rather than full-sign it.
Deterministic	A boolean value that indicates whether the compiler should produce identical assemblies for identical inputs. This parameter corresponds to the `/deterministic` switch of the *vbc.exe* and *csc.exe* compilers.
DisabledWarnings	Suppresses the specified warnings. Only the numeric part of the warning identifier must be specified. Multiple warnings are separated by semicolons. This parameter corresponds to the `/nowarn` switch of the *vbc.exe* compiler.
DisableFastUpToDateCheck	A boolean value that applies to Visual Studio only. The Visual Studio build manager uses a process called FastUpToDateCheck to determine whether a project must be rebuilt to be up to date. This process is faster than using MSBuild to determine this. Setting the DisableFastUpToDateCheck property to `true` lets you bypass the Visual Studio build manager and force it to use MSBuild to determine whether the project is up to date.
DocumentationFile	The name of the file that is generated as the XML documentation file. This name includes only the file name and has no path information.
ErrorReport	Specifies how the compiler task should report internal compiler errors. Valid values are "prompt," "send," or "none." This property is equivalent to the `/errorreport` compiler switch.

PROPERTY OR PARAMETER NAME	DESCRIPTION
ExcludeDeploymentUrl	The GenerateDeploymentManifest task adds a deploymentProvider tag to the deployment manifest if the project file includes any of the following elements: - UpdateUrl - InstallUrl - PublishUrl Using ExcludeDeploymentUrl, however, you can prevent the deploymentProvider tag from being added to the deployment manifest even if any of the above URLs are specified. To do this, add the following property to your project file: `<ExcludeDeploymentUrl>true</ExcludeDeploymentUrl>` **Note:** ExcludeDeploymentUrl is not exposed in the Visual Studio IDE and can be set only by manually editing the project file. Setting this property does not affect publishing within Visual Studio; that is, the deploymentProvider tag will still be added to the URL specified by PublishUrl.
FileAlignment	Specifies, in bytes, where to align the sections of the output file. Valid values are 512, 1024, 2048, 4096, 8192. This property is equivalent to the `/filealignment` compiler switch.
FrameworkPathOverride	Specifies the location of *mscorlib.dll* and *microsoft.visualbasic.dll*. This parameter is equivalent to the `/sdkpath` switch of the *vbc.exe* compiler.
GenerateDocumentation	(C#, Visual Basic) A boolean parameter that indicates whether documentation is generated by the build. If `true`, the build generates documentation information and puts it in an *.xml* file together with the name of the executable file or library that the build task created.
GenerateSerializationAssemblies	Indicates whether XML serialization assemblies should be generated by *SGen.exe*, which can be set to on, auto, or off. This property is used for assemblies that target .NET Framework only. To generate XML serialization assemblies for .NET Standard or .NET Core assemblies, reference the *Microsoft.XmlSerializer.Generator* NuGet package.
IntermediateOutputPath	The full intermediate output path as derived from `BaseIntermediateOutputPath`, if no path is specified. For example, *\obj\debug*.
KeyContainerName	The name of the strong-name key container.
KeyOriginatorFile	The name of the strong-name key file.
MSBuildProjectExtensionsPath	Specifies the path where project extensions are located. By default, this takes the same value as `BaseIntermediateOutputPath`.

PROPERTY OR PARAMETER NAME	DESCRIPTION
ModuleAssemblyName	The name of the assembly that the compiled module is to be incorporated into. The property is equivalent to the `/moduleassemblyname` compiler switch.
NoLogo	A boolean value that indicates whether you want compiler logo to be turned off. This property is equivalent to the `/nologo` compiler switch.
NoStdLib	A boolean value that indicates whether to avoid referencing the standard library (*mscorlib.dll*). The default value is `false` .
NoVBRuntimeReference	A boolean value that indicates whether the Visual Basic runtime (*Microsoft.VisualBasic.dll*) should be included as a reference in the project.
NoWin32Manifest	A boolean value that indicates whether User Account Control (UAC) manifest information will be embedded in the application's executable. Applies only to Visual Studio projects targeting Windows Vista. In projects deployed using ClickOnce and Registration-Free COM, this element is ignored. `False` (the default value) specifies that User Account Control (UAC) manifest information be embedded in the application's executable. `True` specifies that UAC manifest information not be embedded. This property applies only to Visual Studio projects targeting Windows Vista. In projects deployed using ClickOnce and Registration-Free COM, this property is ignored. You should add NoWin32Manifest only if you do not want Visual Studio to embed any manifest information in the application's executable; this process is called *virtualization*. To use virtualization, set `<ApplicationManifest>` in conjunction with `<NoWin32Manifest>` as follows: - For Visual Basic projects, remove the `<ApplicationManifest>` node. (In Visual Basic projects, `<NoWin32Manifest>` is ignored when an `<ApplicationManifest>` node exists.) - For Visual C# projects, set `<ApplicationManifest>` to `False` and `<NoWin32Manifest>` to `True` . (In Visual C# projects, `<ApplicationManifest>` overrides `<NoWin32Manifest>` .) This property is equivalent to the `/nowin32manifest` compiler switch of *vbc.exe*.
Optimize	A boolean value that when set to `true` , enables compiler optimizations. This property is equivalent to the `/optimize` compiler switch.
OptionCompare	Specifies how string comparisons are made. Valid values are "binary" or "text." This property is equivalent to the `/optioncompare` compiler switch of *vbc.exe*.

PROPERTY OR PARAMETER NAME	DESCRIPTION
OptionExplicit	A boolean value that when set to `true`, requires explicit declaration of variables in the source code. This property is equivalent to the `/optionexplicit` compiler switch.
OptionInfer	A boolean value that when set to `true`, enables type inference of variables. This property is equivalent to the `/optioninfer` compiler switch.
OptionStrict	A boolean value that when set to `true`, causes the build task to enforce strict type semantics to restrict implicit type conversions. This property is equivalent to the `/optionstrict` switch of the *vbc.exe* compiler.
OutputPath	Specifies the path to the output directory, relative to the project directory, for example, *bin\Debug*.
OutputType	Specifies the file format of the output file. This parameter can have one of the following values: - Library. Creates a code library. (Default value.) - Exe. Creates a console application. - Module. Creates a module. - Winexe. Creates a Windows-based program. This property is equivalent to the `/target` switch of the *vbc.exe* compiler.
OverwriteReadOnlyFiles	A boolean value that indicates whether you want to enable the build to overwrite read-only files or trigger an error.
PathMap	Specifies how to map physical paths to source path names output by the compiler. This property is equivalent to the `/pathmap` switch of the *csc.exe* compiler.
PdbFile	The file name of the *.pdb* file that you are emitting. This property is equivalent to the `/pdb` switch of the *csc.exe* compiler.
Platform	The operating system you are building for. Valid values are "Any CPU", "x86", and "x64".
ProduceReferenceAssembly	A boolean value that when set to `true` enables production of reference assemblies for the current assembly. `Deterministic` should be `true` when using this feature. This property corresponds to the `/refout` switch of the *vbc.exe* and *csc.exe* compilers.
ProduceOnlyReferenceAssembly	A boolean value that instructs the compiler to emit only a reference assembly rather than compiled code. Cannot be used in conjunction with `ProduceReferenceAssembly`. This property corresponds to the `/refonly` switch of the *vbc.exe* and *csc.exe* compilers.

PROPERTY OR PARAMETER NAME	DESCRIPTION
RemoveIntegerChecks	A boolean value that indicates whether to disable integer overflow error checks. The default value is `false`. This property is equivalent to the `/removeintchecks` switch of the *vbc.exe* compiler.
SGenUseProxyTypes	A boolean value that indicates whether proxy types should be generated by *SGen.exe*. This applies only when *GenerateSerializationAssemblies* is set to on, and for .NET Framework only. The SGen target uses this property to set the UseProxyTypes flag. This property defaults to true, and there is no UI to change this. To generate the serialization assembly for non-webservice types, add this property to the project file and set it to false before importing the *Microsoft.Common.Targets* or the *C#/VB.targets*.
SGenToolPath	An optional tool path that indicates where to obtain *SGen.exe* when the current version of *SGen.exe* is overridden. This property is used for .NET Framework only.
StartupObject	Specifies the class or module that contains the Main method or Sub Main procedure. This property is equivalent to the `/main` compiler switch.
ProcessorArchitecture	The processor architecture that is used when assembly references are resolved. Valid values are "msil," "x86," "amd64," or "ia64."
RootNamespace	The root namespace to use when you name an embedded resource. This namespace is part of the embedded resource manifest name.
Satellite_AlgorithmId	The ID of the *ALexe* hashing algorithm to use when satellite assemblies are created.
Satellite_BaseAddress	The base address to use when culture-specific satellite assemblies are built by using the `CreateSatelliteAssemblies` target.
Satellite_CompanyName	The company name to pass into *ALexe* during satellite assembly generation.
Satellite_Configuration	The configuration name to pass into *ALexe* during satellite assembly generation.
Satellite_Description	The description text to pass into *ALexe* during satellite assembly generation.
Satellite_EvidenceFile	Embeds the specified file in the satellite assembly that has the resource name "Security.Evidence."
Satellite_FileVersion	Specifies a string for the File Version field in the satellite assembly.

PROPERTY OR PARAMETER NAME	DESCRIPTION
Satellite_Flags	Specifies a value for the Flags field in the satellite assembly.
Satellite_GenerateFullPaths	Causes the build task to use absolute paths for any files reported in an error message.
Satellite_LinkResource	Links the specified resource files to a satellite assembly.
Satellite_MainEntryPoint	Specifies the fully-qualified name (that is, class.method) of the method to use as an entry point when a module is converted to an executable file during satellite assembly generation.
Satellite_ProductName	Specifies a string for the Product field in the satellite assembly.
Satellite_ProductVersion	Specifies a string for the ProductVersion field in the satellite assembly.
Satellite_TargetType	Specifies the file format of the satellite assembly output file as "library," "exe," or "win." The default value is "library."
Satellite_Title	Specifies a string for the Title field in the satellite assembly.
Satellite_Trademark	Specifies a string for the Trademark field in the satellite assembly.
Satellite_Version	Specifies the version information for the satellite assembly.
Satellite_Win32Icon	Inserts an *.ico* icon file in the satellite assembly.
Satellite_Win32Resource	Inserts a Win32 resource (*.res* file) into the satellite assembly.
SubsystemVersion	Specifies the minimum version of the subsystem that the generated executable file can use. This property is equivalent to the `/subsystemversion` compiler switch. For information about the default value of this property, see /subsystemversion (Visual Basic) or /subsystemversion (C# compiler options).
TargetCompactFramework	The version of the .NET Compact Framework that is required to run the application that you are building. Specifying this lets you reference certain framework assemblies that you may not be able to reference otherwise.
TargetFrameworkVersion	The version of the .NET Framework that is required to run the application that you are building. Specifying this lets you reference certain framework assemblies that you may not be able to reference otherwise.
TreatWarningsAsErrors	A boolean parameter that, if `true`, causes all warnings to be treated as errors. This parameter is equivalent to the `/nowarn` compiler switch.
UseHostCompilerIfAvailable	A boolean parameter that, if `true`, causes the build task to use the in-process compiler object, if it is available. This parameter is used only by Visual Studio.

PROPERTY OR PARAMETER NAME	DESCRIPTION
Utf8Output	A boolean parameter that, if `true`, logs compiler output by using UTF-8 encoding. This parameter is equivalent to the `/utf8Output` compiler switch.
VbcToolPath	An optional path that indicates another location for *vbc.exe* when the current version of *vbc.exe* is overridden.
VbcVerbosity	Specifies the verbosity of the Visual Basic compiler's output. Valid values are "Quiet," "Normal" (the default value), or "Verbose."
VisualStudioVersion	Specifies the version of Visual Studio under which this project should be considered to be running. If this property isn't specified, MSBuild sets it to a reasonable default value. This property is used in several project types to specify the set of targets that are used for the build. If `ToolsVersion` is set to 4.0 or higher for a project, `VisualStudioVersion` is used to specify which sub-toolset to use. For more information, see Toolset (ToolsVersion).
WarningsAsErrors	Specifies a list of warnings to treat as errors. This parameter is equivalent to the `/warnaserror` compiler switch.
WarningsNotAsErrors	Specifies a list of warnings that are not treated as errors. This parameter is equivalent to the `/warnaserror` compiler switch.
Win32Manifest	The name of the manifest file that should be embedded in the final assembly. This parameter is equivalent to the `/win32Manifest` compiler switch.
Win32Resource	The file name of the Win32 resource to be embedded in the final assembly. This parameter is equivalent to the `/win32resource` compiler switch.

See also

- Common MSBuild project items

Common MSBuild project items

11/12/2019 • 5 minutes to read • Edit Online

In MSBuild, an item is a named reference to one or more files. Items contain metadata such as file names, paths, and version numbers. All project types in Visual Studio have several items in common. These items are defined in the file *Microsoft.Build.CommonTypes.xsd*.

Common items

The following is a list of all the common project items.

Reference

Represents an assembly (managed) reference in the project.

ITEM METADATA NAME	DESCRIPTION
HintPath	Optional string. Relative or absolute path of the assembly.
Name	Optional string. The display name of the assembly, for example, "System.Windows.Forms."
FusionName	Optional string. Specifies the simple or strong fusion name for the item. When this attribute is present, it can save time because the assembly file does not have to be opened to obtain the fusion name.
SpecificVersion	Optional boolean. Specifies whether only the version in the fusion name should be referenced.
Aliases	Optional string. Any aliases for the reference.
Private	Optional boolean. Specifies whether the reference should be copied to the output folder. This attribute matches the **Copy Local** property of the reference that's in the Visual Studio IDE.

COMReference

Represents a COM (unmanaged) component reference in the project. This item applies only to .NET projects.

ITEM METADATA NAME	DESCRIPTION
Name	Optional string. The display name of the component.
Guid	Required string. A GUID for the component, in the form {12345678-1234-1234-1234-1234567891234}.
VersionMajor	Required string. The major part of the version number of the component. For example, "5" if the full version number is "5.46."

ITEM METADATA NAME	DESCRIPTION
VersionMinor	Required string. The minor part of the version number of the component. For example, "46" if the full version number is "5.46."
LCID	Optional string. The LocaleID for the component.
WrapperTool	Optional string. The name of the wrapper tool that is used on the component, for example, "tlbimp."
Isolated	Optional boolean. Specifies whether the component is a reg-free component.

COMFileReference

Represents a list of type libraries that are passed to the `TypeLibFiles` parameter of the ResolveComReference target. This item applies only to .NET projects.

ITEM METADATA NAME	DESCRIPTION
WrapperTool	Optional string. The name of the wrapper tool that is used on the component, for example, "tlbimp."

NativeReference

Represents a native manifest file or a reference to such a file.

ITEM METADATA NAME	DESCRIPTION
Name	Required string. The base name of the manifest file.
HintPath	Required string. The relative path of the manifest file.

ProjectReference

Represents a reference to another project.

ITEM METADATA NAME	DESCRIPTION
Name	Optional string. The display name of the reference.
Project	Optional string. A GUID for the reference, in the form {12345678-1234-1234-1234-1234567891234}.
Package	Optional string. The path of the project file that is being referenced.
ReferenceOutputAssembly	Optional boolean. If set to `false`, does not include the output of the referenced project as a Reference of this project, but still ensures that the other project builds before this one. Defaults to `true`.

Compile

Represents the source files for the compiler.

ITEM METADATA NAME	DESCRIPTION
DependentUpon	Optional string. Specifies the file this file depends on to compile correctly.
AutoGen	Optional boolean. Indicates whether the file was generated for the project by the Visual Studio integrated development environment (IDE).
Link	Optional string. The notational path to be displayed when the file is physically located outside the influence of the project file.
Visible	Optional boolean. Indicates whether to display the file in **Solution Explorer** in Visual Studio.
CopyToOutputDirectory	Optional string. Determines whether to copy the file to the output directory. Values are: 1. Never 2. Always 3. PreserveNewest

EmbeddedResource

Represents resources to be embedded in the generated assembly.

ITEM METADATA NAME	DESCRIPTION
DependentUpon	Optional string. Specifies the file this file depends on to compile correctly
Generator	Required string. The name of any file generator that is run on this item.
LastGenOutput	Required string. The name of the file that was created by any file generator that ran on this item.
CustomToolNamespace	Required string. The namespace in which any file generator that runs on this item should create code.
Link	Optional string. The notational path is displayed if the file is physically located outside the influence of the project.
Visible	Optional boolean. Indicates whether to display the file in **Solution Explorer** in Visual Studio.
CopyToOutputDirectory	Optional string. Determines whether to copy the file to the output directory. Values are: 1. Never 2. Always 3. PreserveNewest
LogicalName	Required string. The logical name of the embedded resource.

Content

Represents files that are not compiled into the project, but may be embedded or published together with it.

ITEM METADATA NAME	DESCRIPTION
DependentUpon	Optional string. Specifies the file this file depends on to compile correctly.
Generator	Required string. The name of any file generator that runs on this item.
LastGenOutput	Required string. The name of the file that was created by any file generator that was run on this item.
CustomToolNamespace	Required string. The namespace in which any file generator that runs on this item should create code.
Link	Optional string. The notational path to be displayed if the file is physically located outside the influence of the project.
PublishState	Required string. The publish state of the content, either: - Default - Included - Excluded - DataFile - Prerequisite
IsAssembly	Optional boolean. Specifies whether the file is an assembly.
Visible	Optional boolean. Indicates whether to display the file in **Solution Explorer** in Visual Studio.
CopyToOutputDirectory	Optional string. Determines whether to copy the file to the output directory. Values are: 1. Never 2. Always 3. PreserveNewest

None

Represents files that should have no role in the build process.

ITEM METADATA NAME	DESCRIPTION
DependentUpon	Optional string. Specifies the file this file depends on to compile correctly.
Generator	Required string. The name of any file generator that is run on this item.
LastGenOutput	Required string. The name of the file that was created by any file generator that ran on this item.
CustomToolNamespace	Required string. The namespace in which any file generator that runs on this item should create code.

ITEM METADATA NAME	DESCRIPTION
Link	Optional string. The notational path to be displayed if the file is physically located outside the influence of the project.
Visible	Optional boolean. Indicates whether to display the file in **Solution Explorer** in Visual Studio.
CopyToOutputDirectory	Optional string. Determines whether to copy the file to the output directory. Values are: 1. Never 2. Always 3. PreserveNewest

AssemblyMetadata

Represents assembly attributes to be generated as `[AssemblyMetadata(key, value)]`.

ITEM METADATA NAME	DESCRIPTION
Include	Becomes the first parameter (the key) in the `AssemblyMetadataAttribute` attribute constructor.
Value	Required string. Becomes the second parameter (the value) in the `AssemblyMetadataAttribute` attribute constructor.

> **NOTE**
> This applies to projects using the .NET Core SDK only.

BaseApplicationManifest

Represents the base application manifest for the build, and contains ClickOnce deployment security information.

CodeAnalysisImport

Represents the FxCop project to import.

Import

Represents assemblies whose namespaces should be imported by the Visual Basic compiler.

See also

- Common MSBuild project properties

MSBuild command-line reference

10/10/2019 • 9 minutes to read • Edit Online

When you use *MSBuild.exe* to build a project or solution file, you can include several switches to specify various aspects of the process.

Every switch is available in two forms: -switch and /switch. The documentation only shows the -switch form.

Syntax

```
MSBuild.exe [Switches] [ProjectFile]
```

Arguments

ARGUMENT	DESCRIPTION
`ProjectFile`	Builds the targets in the project file that you specify. If you don't specify a project file, MSBuild searches the current working directory for a file name extension that ends in *proj* and uses that file. You can also specify a Visual Studio solution file for this argument.

Switches

SWITCH	SHORT FORM	DESCRIPTION
-help	/? or -h	Display usage information. The following command is an example: `msbuild.exe -?`
-detailedsummary	-ds	Show detailed information at the end of the build log about the configurations that were built and how they were scheduled to nodes.
-ignoreprojectextensions: `extensions`	-ignore: `extensions`	Ignore the specified extensions when determining which project file to build. Use a semicolon or a comma to separate multiple extensions, as the following example shows: `-ignoreprojectextensions:.vcproj,.sln`
-maxcpucount[: `number`]	-m[: `number`]	Specifies the maximum number of concurrent processes to use when building. If you don't include this switch, the default value is 1. If you include this switch without specifying a value, MSBuild will use up to the number of processors in the computer. For more information, see Building multiple projects in parallel. The following example instructs MSBuild to build using three MSBuild processes, which allows three projects to build at the same time: `msbuild myproject.proj -maxcpucount:3`
-noautoresponse	-noautorsp	Don't include any *MSBuild.rsp* files automatically.

SWITCH	SHORT FORM	DESCRIPTION
-nodeReuse: `value`	-nr: `value`	Enable or disable the re-use of MSBuild nodes. You can specify the following values: - **True**. Nodes remain after the build finishes so that subsequent builds can use them (default). - **False**. Nodes don't remain after the build completes. A node corresponds to a project that's executing. If you include the **-maxcpucount** switch, multiple nodes can execute concurrently.
-nologo		Don't display the startup banner or the copyright message.
-preprocess[: `filepath`]	-pp[: `filepath`]	Create a single, aggregated project file by inlining all the files that would be imported during a build, with their boundaries marked. You can use this switch to more easily determine which files are being imported, from where the files are being imported, and which files contribute to the build. When you use this switch, the project isn't built. If you specify a `filepath`, the aggregated project file is output to the file. Otherwise, the output appears in the console window. For information about how to use the `Import` element to insert a project file into another project file, see Import element (MSBuild) and How to: Use the same target in multiple project files.
-property: `name` = `value`	-p: `name` = `value`	Set or override the specified project-level properties, where `name` is the property name and `value` is the property value. Specify each property separately, or use a semicolon or comma to separate multiple properties, as the following example shows: `-property:WarningLevel=2;OutDir=bin\Debug`
-restore	-r	Runs the `Restore` target prior to building the actual targets.
-target: `targets`	-t: `targets`	Build the specified targets in the project. Specify each target separately, or use a semicolon or comma to separate multiple targets, as the following example shows: `-target:PrepareResources;Compile` If you specify any targets by using this switch, they are run instead of any targets in the `DefaultTargets` attribute in the project file. For more information, see Target build order and How to: Specify which target to build first. A target is a group of tasks. For more information, see Targets.

SWITCH	SHORT FORM	DESCRIPTION
-toolsversion: `version`	-tv: `version`	Specifies the version of the Toolset to use to build the project, as the following example shows: `-toolsversion:3.5` By using this switch, you can build a project and specify a version that differs from the version that's specified in the Project element (MSBuild). For more information, see Overriding ToolsVersion settings. For MSBuild 4.5, you can specify the following values for `version`: 2.0, 3.5, and 4.0. If you specify 4.0, the `VisualStudioVersion` build property specifies which sub-toolset to use. For more information, see the Sub-toolsets section of Toolset (ToolsVersion). A Toolset consists of tasks, targets, and tools that are used to build an application. The tools include compilers such as *csc.exe* and *vbc.exe*. For more information about Toolsets, see Toolset (ToolsVersion), Standard and custom toolset configurations, and Multitargeting. **Note:** The toolset version isn't the same as the target framework, which is the version of the .NET Framework on which a project is built to run. For more information, see Target framework and target platform.
-validate:[`schema`]	-val[`schema`]	Validate the project file and, if validation succeeds, build the project. If you don't specify `schema`, the project is validated against the default schema. If you specify `schema`, the project is validated against the schema that you specify. The following setting is an example: `-validate:MyExtendedBuildSchema.xsd`
-verbosity: `level`	-v: `level`	Specifies the amount of information to display in the build log. Each logger displays events based on the verbosity level that you set for that logger. You can specify the following verbosity levels: `q[uiet]`, `m[inimal]`, `n[ormal]`, `d[etailed]`, and `diag[nostic]`. The following setting is an example: `-verbosity:quiet`
-version	-ver	Display version information only. The project isn't built.
@ `file`		Insert command-line switches from a text file. If you have multiple files, you specify them separately. For more information, see Response files.

Switches for loggers

SWITCH	SHORT FORM	DESCRIPTION
-consoleloggerparameters: `parameters`	-clp: `parameters`	Pass the parameters that you specify to the console logger, which displays build information in the console window. You can specify the following parameters: - **PerformanceSummary**. Show the time that's spent in tasks, targets, and projects. - **Summary**. Show the error and warning summary at the end. - **NoSummary**. Don't show the error and warning summary at the end. - **ErrorsOnly**. Show only errors. - **WarningsOnly**. Show only warnings. - **NoItemAndPropertyList**. Don't show the list of items and properties that would appear at the start of each project build if the verbosity level is set to `diagnostic`. - **ShowCommandLine**. Show `TaskCommandLineEvent` messages. - **ShowTimestamp**. Show the timestamp as a prefix to any message. - **ShowEventId**. Show the event ID for each started event, finished event, and message. - **ForceNoAlign**. Don't align the text to the size of the console buffer. - **DisableConsoleColor**. Use the default console colors for all logging messages. - **DisableMPLogging**. Disable the multiprocessor logging style of output when running in non-multiprocessor mode. - **EnableMPLogging**. Enable the multiprocessor logging style even when running in non-multiprocessor mode. This logging style is on by default. - **Verbosity**. Override the **-verbosity** setting for this logger. Use a semicolon to separate multiple parameters, as the following example shows: `-consoleloggerparameters:PerformanceSummary;NoSu -verbosity:minimal`
-distributedFileLogger	-dfl	Log the build output of each MSBuild node to its own file. The initial location for these files is the current directory. By default, the files are named *MSBuild<NodeId>.log*. You can use the **-fileLoggerParameters** switch to specify the location of the files and other parameters for the fileLogger. If you name a log file by using the **-fileLoggerParameters** switch, the distributed logger will use that name as a template and append the node ID to that name when creating a log file for each node.

SWITCH	SHORT FORM	DESCRIPTION

-distributedlogger:
 	
`central logger` *
 	
`forwarding logger`
	-dl: `central logger` * `forwarding logger`	Log events from MSBuild, attaching a different logger instance to each node. To specify multiple loggers, specify each logger separately.
		
		You use the logger syntax to specify a logger. For the logger syntax, see the **-logger** switch below.
		
		The following examples show how to use this switch:
		
		`-dl:XMLLogger,MyLogger,Version=1.0.2,Culture=neu`
		
		`-dl:MyLogger,C:\My.dll*ForwardingLogger,C:\Logge`

-fileLogger *[number]*	-fl[`number`]	Log the build output to a single file in the current directory. If you don't specify `number`, the output file is named *msbuild.log*. If you specify `number`, the output file is named *msbuild<n>.log*, where <n> is `number`. `Number` can be a digit from 1 to 9. You can use the **-fileLoggerParameters** switch to specify the location of the file and other parameters for the fileLogger.

SWITCH	SHORT FORM	DESCRIPTION

-fileloggerparameters:[number]
 	
`parameters`
	-flp:[`number`] `parameters`	Specifies any extra parameters for the file logger and the distributed file logger. The presence of this switch implies that the corresponding **-filelogger[** `number` **]** switch is present. `Number` can be a digit from 1 to 9.
		
		You can use all parameters that are listed for **-consoleloggerparameters**. You can also use one or more of the following parameters:
		
		- **LogFile**. The path to the log file into which the build log is written. The distributed file logger prefixes this path to the names of its log files.
		- **Append**. Determines whether the build log is appended to the log file or overwrites it. When you set the switch, the build log is appended to the log file. When the switch is not present, the contents of an existing log file are overwritten.
		If you include the append switch, no matter whether it is set to true or false, the log is appended. If you do not include the append switch, the log is overwritten.
		In this case the file is overwritten:
		`msbuild myfile.proj -l:FileLogger,Microsoft.Build;logfile=MyLog.log`
		In this case the file is appended:
		`msbuild myfile.proj -l:FileLogger,Microsoft.Build;logfile=MyLog.log;`
		In this case the file is appended:
		`msbuild myfile.proj -l:FileLogger,Microsoft.Build;logfile=MyLog.log;`
		- **Encoding**. Specifies the encoding for the file (for example, UTF-8, Unicode, or ASCII).
		
		The following example generates separate log files for warnings and errors:
		
		`-flp1:logfile=errors.txt;errorsonly -flp2:logfile=warnings.txt;warningsonly`
		
		The following examples show other possibilities:
		
		`-fileLoggerParameters:LogFile=MyLog.log;Append;Verbosity=diagnostic;Encoding=UTF-8`
		
		`-flp:Summary;Verbosity=minimal;LogFile=msbuild.s`
		
		`-flp1:warningsonly;logfile=msbuild.wrn`
		
		`-flp2:errorsonly;logfile=msbuild.err`

SWITCH	SHORT FORM	DESCRIPTION
-binaryLogger[:[LogFile=]`output.binlog`[;ProjectImports=[None,Embed,ZipFile]]]	-bl	Serializes all build events to a compressed binary file. By default the file is in the current directory and named *msbuild.binlog*. The binary log is a detailed description of the build process that can later be used to reconstruct text logs and used by other analysis tools. A binary log is usually 10-20x smaller than the most detailed text diagnostic-level log, but it contains more information. The binary logger by default collects the source text of project files, including all imported projects and target files encountered during the build. The optional `ProjectImports` switch controls this behavior: - **ProjectImports=None**. Don't collect the project imports. - **ProjectImports=Embed**. Embed project imports in the log file (default). - **ProjectImports=ZipFile**. Save project files to *<output>.projectimports.zip* where <output> is the same name as the binary log file name. The default setting for ProjectImports is Embed. **Note**: the logger does not collect non-MSBuild source files such as *.cs*, *.cpp* etc. A *.binlog* file can be "played back" by passing it to *msbuild.exe* as an argument instead of a project/solution. Other loggers will receive the information contained in the log file as if the original build was happening. You can read more about the binary log and its usages at: https://github.com/Microsoft/msbuild/wiki/Binary-Log **Examples**: - `-bl` - `-bl:output.binlog` - `-bl:output.binlog;ProjectImports=None` - `-bl:output.binlog;ProjectImports=ZipFile` - `-bl:..\..\custom.binlog` - `-binaryLogger`

SWITCH	SHORT FORM	DESCRIPTION
-logger: `logger`	-l: `logger`	Specifies the logger to use to log events from MSBuild. To specify multiple loggers, specify each logger separately. Use the following syntax for `logger`: `[``LoggerClass``,]``LoggerAssembly``[;``LoggerP`  Use the following syntax for `LoggerClass`: `[``PartialOrFullNamespace``.]``LoggerClassName`  You don't have to specify the logger class if the assembly contains exactly one logger.  Use the following syntax for `LoggerAssembly`: `{``AssemblyName``[,``StrongName``]` `|` `AssemblyFile``}`  Logger parameters are optional and are passed to the logger exactly as you enter them.  The following examples use the **-logger** switch.  `-logger:XMLLogger,MyLogger,Version=1.0.2,Culture`  `-logger:XMLLogger,C:\Loggers\MyLogger.dll;Output`
-noconsolelogger	-noconlog	Disable the default console logger, and don't log events to the console.

Example

The following example builds the `rebuild` target of the *MyProject.proj* project.

```
MSBuild.exe MyProject.proj -t:rebuild
```

Example

You can use *MSBuild.exe* to perform more complex builds. For example, you can use it to build specific targets of specific projects in a solution. The following example rebuilds the project `NotInSolutionFolder` and cleans the project `InSolutionFolder`, which is in the *NewFolder* solution folder.

```
msbuild SlnFolders.sln -t:NotInSolutionfolder:Rebuild;NewFolder\InSolutionFolder:Clean
```

See also

- MSBuild reference
- Common MSBuild project properties

MSBuild .targets files

11/13/2019 • 2 minutes to read • Edit Online

MSBuild includes several *.targets* files that contain items, properties, targets, and tasks for common scenarios. These files are automatically imported into most Visual Studio project files to simplify maintenance and readability.

Projects typically import one or more *.targets* files to define their build process. For example a Visual C# project created by Visual Studio will import *Microsoft.CSharp.targets* which imports *Microsoft.Common.targets*. The Visual C# project itself will define the items and properties specific to that project, but the standard build rules for a Visual C# project are defined in the imported *.targets* files.

The `$(MSBuildToolsPath)` value specifies the path of these common *.targets* files. If the `ToolsVersion` is 4.0, the files are in the following location: *<WindowsInstallationPath>\Microsoft.NET\Framework\v4.0.30319*

> **NOTE**
> For information about how to create your own targets, see Targets. For information about how to use the `Import` element to insert a project file into another project file, see Import element (MSBuild) and How to: Use the same target in multiple project files.

Common .targets files

.TARGETS FILE	DESCRIPTION
Microsoft.Common.targets	Defines the steps in the standard build process for Visual Basic and Visual C# projects. Imported by the *Microsoft.CSharp.targets* and *Microsoft.VisualBasic.targets* files, which include the following statement: `<Import Project="Microsoft.Common.targets" />`
Microsoft.CSharp.targets	Defines the steps in the standard build process for Visual C# projects. Imported by Visual C# project files (*.csproj*), which include the following statement: `<Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />`
Microsoft.VisualBasic.targets	Defines the steps in the standard build process for Visual Basic projects. Imported by Visual Basic project files (*.vbproj*), which include the following statement: `<Import Project="$(MSBuildToolsPath)\Microsoft.VisualBasic.targets" />`

Directory.Build.targets

Directory.Build.targets is a user-defined file that provides customizations to projects under a directory. This file is automatically imported from *Microsoft.Common.targets* unless the property **ImportDirectoryBuildTargets** is set to **false**. For more information, Customize your build.

See also

- Import element (MSBuild)
- MSBuild reference
- MSBuild

MSBuild well-known item metadata

2/21/2019 • 2 minutes to read • Edit Online

The following table describes the metadata assigned to every item upon creation. In each example, the following item declaration was used to include the file *C:\MyProject\Source\Program.cs* in the project.

```
<ItemGroup>
    <MyItem Include="Source\Program.cs" />
</ItemGroup>
```

ITEM METADATA	DESCRIPTION
%(FullPath)	Contains the full path of the item. For example: *C:\MyProject\Source\Program.cs*
%(RootDir)	Contains the root directory of the item. For example: *C:*
%(Filename)	Contains the file name of the item, without the extension. For example: *Program*
%(Extension)	Contains the file name extension of the item. For example: *.cs*
%(RelativeDir)	Contains the path specified in the `Include` attribute, up to the final backslash (\). For example: *Source*
%(Directory)	Contains the directory of the item, without the root directory. For example: *MyProject\Source*

ITEM METADATA	DESCRIPTION

%(RecursiveDir)	If the `Include` attribute contains the wildcard **, this metadata specifies the part of the path that replaces the wildcard. For more information on wildcards, see How to: Select the files to build.
	
	If the folder *C:\MySolution\MyProject\Source* contains the file *Program.cs*, and if the project file contains this item:
	
	`<ItemGroup>`
	
	`<MyItem Include="C:\**\Program.cs" />`
	
	`</ItemGroup>`
	
	then the value of `%(MyItem.RecursiveDir)` would be *MySolution\MyProject\Source*.

%(Identity)	The item specified in the `Include` attribute.. For example: *Source\Program.cs*
%(ModifiedTime)	Contains the timestamp from the last time the item was modified. For example: `2004-07-01 00:21:31.5073316`
%(CreatedTime)	Contains the timestamp from when the item was created. For example: `2004-06-25 09:26:45.8237425`
%(AccessedTime)	Contains the timestamp from the last time the item was accessed. `2004-08-14 16:52:36.3168743`

See also

- Items
- Batching
- MSBuild reference

MSBuild response files

2/21/2019 • 2 minutes to read • Edit Online

Response (*.rsp*) files are text files that contain *MSBuild.exe* command-line switches. Each switch can be on a separate line or all switches can be on one line. Comment lines are prefaced with a **#** symbol. The **@** switch is used to pass another response file to *MSBuild.exe*.

MSBuild.rsp

The autoresponse file is a special *.rsp* file that *MSBuild.exe* automatically uses when building a project. This file, *MSBuild.rsp*, must be in the same directory as *MSBuild.exe*, otherwise it will not be found. You can edit this file to specify default command-line switches to *MSBuild.exe*. For example, if you use the same logger every time you build a project, you can add the **-logger** switch to *MSBuild.rsp*, and *MSBuild.exe* will use the logger every time a project is built.

Directory.Build.rsp

In version 15.6 and above, MSBuild will search parent directories of the project for a file named *Directory.Build.rsp*. This can be helpful in a source code repository to provide default arguments during command-line builds. It can also be used to specify the command-line arguments of hosted builds.

See also

- MSBuild reference
- Command-line reference

WPF MSBuild reference

2/21/2019 • 2 minutes to read • Edit Online

Windows Presentation Foundation (WPF) extends Microsoft build engine (MSBuild) with additional build support, which is documented in this section.

In this section

WPF .targets files

Describes WPF .targets files.

WPF MSBuild task reference

Lists the available WPF build tasks.

Microsoft.Build.Tasks

A build task assembly.

Microsoft.Build.Tasks.Deployment.Bootstrapper

A build task deployment bootstrapper assembly.

Microsoft.Build.Tasks.Deployment.ManifestUtilities

A build task deployment manifest utility assembly.

Microsoft.Build.Tasks.Hosting

A build task hosting assembly.

Microsoft.Build.Tasks.Windows

A build task windows assembly.

See also

- MSBuild

WPF .targets files

2/21/2019 • 2 minutes to read • Edit Online

Windows Presentation Foundation (WPF) extends the Microsoft build engine (MSBuild) by adding a set of WPF-specific tasks that are combined into a special *.targets* file, *Microsoft.WinFX.targets*. This file combines the set of MSBuild tasks that are required to build an MSBuild project in Windows Presentation Foundation (WPF).

See also

- MSBuild .targets files
- MSBuild reference
- Building a WPF application (WPF)

WPF MSBuild task reference

2/21/2019 • 2 minutes to read • Edit Online

The Windows Presentation Foundation (WPF) build process extends Microsoft build engine (MSBuild) with an additional set of build tasks, including tasks to compile markup and process resources.

In this section

- FileClassifier

Classifies a set of source resources as those that will be embedded into an assembly. If a resource is not localizable, it is embedded into the main application assembly; otherwise, it is embedded into a satellite assembly.

- GenerateTemporaryTargetAssembly

Generates an assembly if at least one Extensible Application Markup Language (XAML) page in a project references a type that is declared locally in that project. The generated assembly is removed after the build process is completed, or if the build process fails.

- GetWinFXPath

Returns the directory of the current Microsoft .NET Framework runtime.

- MarkupCompilePass1

Converts non-localizable Extensible Application Markup Language (XAML) project files to compiled binary format.

- MarkupCompilePass2

Performs second-pass markup compilation on Extensible Application Markup Language (XAML) files that reference types in the same project.

- MergeLocalizationDirectives

Merges the localization attributes and comments of one or more XAML binary format files into a single file for the whole assembly.

- ResourcesGenerator

Embeds one or more resources (*.jpg*, *.ico*, *.bmp*, XAML in binary format, and other extension types) into a *.resources* file.

- UidManager

Checks, updates, or removes unique identifiers (UIDs), in order to localize all Extensible Application Markup Language (XAML) elements that are included in the source XAML files.

- UpdateManifestForBrowserApplication

Adds the **<hostInBrowser />** element to the application manifest (<*projectname*>*.exe.manifest*) when a XAML browser application (XBAP) project is built.

See also

- MSBuild

FileClassifier task

2/21/2019 • 2 minutes to read • Edit Online

The FileClassifier task classifies a set of source resources as those that will be embedded into an assembly. If a resource is not localizable, it is embedded into the main application assembly; otherwise, it is embedded into a satellite assembly.

Task parameters

PARAMETER	DESCRIPTION
`CLREmbeddedResource`	Unused.
`CLRResourceFiles`	Unused.
`CLRSatelliteEmbeddedResource`	Unused.
`Culture`	Optional **String** parameter. Specifies the culture for the build. This value can be **null** if the build is non-localizable. If **null**, the default value is the lowercase value that **CultureInfo.InvariantCulture** returns.
`MainEmbeddedFiles`	Optional **ITaskItem[]** output parameter. Specifies the non-localizable resources that are embedded into the main assembly.
`OutputType`	Required **String** parameter. Specifies the type of file to embed the specified source files into. The valid values are **exe**, **winexe**, or **library**.
`SatelliteEmbeddedFiles`	Optional **ITaskItem[]** output parameter. Specifies the localizable files that are embedded into the satellite assembly for the culture specified by the **Culture** parameter.
`SourceFiles`	Required **ITaskItem[]** parameter. Specifies the list of files to classify.

Remarks

If the **Culture** parameter is not set, all resources that are specified by using the **SourceFiles** parameter are non-localizable; otherwise, they are localizable, unless they are associated with a **Localizable** attribute that is set to **false**.

Example

The following example classifies a single source file as a resource and then embeds it in a satellite assembly for the

French-Canadian (fr-CA) culture.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <UsingTask
    TaskName="Microsoft.Build.Tasks.Windows.FileClassifier"
    AssemblyFile="C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0\PresentationBuildTasks.dll"
/>
  <ItemGroup>
    <Resource Include="Resource1.bmp" />
  </ItemGroup>
  <Target Name="FileClassifierTask">
    <FileClassifier
      SourceFiles="Resource1.bmp"
      Culture="fr-CA"
      OutputType="exe" />
  </Target>
</Project>
```

See also

- WPF MSBuild reference
- Task reference
- MSBuild reference
- Task reference
- Build a WPF application (WPF)

GenerateTemporaryTargetAssembly task

2/21/2019 • 2 minutes to read • Edit Online

The GenerateTemporaryTargetAssembly task generates an assembly if at least one Extensible Application Markup Language (XAML) page in a project references a type that is declared locally in that project. The generated assembly is removed after the build process is completed, or if the build process fails.

Task parameters

PARAMETER	DESCRIPTION
`AssemblyName`	Required **String** parameter. Specifies the short name of the assembly that is generated for a project and is also the name of the target assembly that is temporarily generated. For example, if a project generates a Windows executable whose name is *WinExeAssembly.exe*, the **AssemblyName** parameter has a value of **WinExeAssembly**.
`CompileTargetName`	Required **String** parameter. Specifies the name of the Microsoft build engine (MSBuild) target that is used to generate assemblies from source code files. The typical value for **CompileTargetName** is **CoreCompile**.
`CompileTypeName`	Required **String** parameter. Specifies the type of compilation that is performed by the target that is specified by the **CompileTargetName** parameter. For the **CoreCompile** target, this value is **Compile**.
`CurrentProject`	Required **String** parameter. Specifies the full path of the MSBuild project file for the project that requires a temporary target assembly.
`GeneratedCodeFiles`	Optional **ITaskItem[]** parameter. Specifies the list of language-specific managed code files that were generated by the MarkupCompilePass1 task.
`IntermediateOutputPath`	Required **String** parameter. Specifies the directory that the temporary target assembly is generated to.
`MSBuildBinPath`	Required **String** parameter. Specifies the location of *MSBuild.exe*, which is required to compile the temporary target assembly.

PARAMETER	DESCRIPTION
`ReferencePath`	Optional **ITaskItem[]** parameter. Specifies a list of assemblies, by path and file name, that are referenced by the types that are compiled into the temporary target assembly.
`ReferencePathTypeName`	Required **String** parameter. Specifies the parameter that is used by the compilation target (**CompileTargetName**) parameter that specifies the list of assembly references (**ReferencePath**). The appropriate value is **ReferencePath**.

Remarks

The first markup compilation pass, which is run by the MarkupCompilePass1, compiles XAML files to binary format. Consequently, the compiler needs a list of the referenced assemblies that contain the types that are used by the XAML files. However, if a XAML file uses a type that is defined in the same project, a corresponding assembly for that project is not created until the project is built. Therefore, an assembly reference cannot be provided during the first markup compilation pass.

Instead, **MarkupCompilePass1** defers the conversion of XAML files that contain references to types in the same project to a second markup compilation pass, which is executed by the MarkupCompilePass2. Before **MarkupCompilePass2** is executed, a temporary assembly is generated. This assembly contains the types that are used by the XAML files whose markup compilation pass was deferred. A reference to the generated assembly is provided to **MarkupCompilePass2** when it runs to allow the deferred compilation XAML files to be converted to binary format.

Example

The following example generates a temporary assembly because *Page1.xaml* contains a reference to a type that is in the same project.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <UsingTask
    TaskName="Microsoft.Build.Tasks.Windows.GenerateTemporaryTargetAssembly"
    AssemblyFile="C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0\PresentationBuildTasks.dll"
/>
  <Target Name="GenerateTemporaryTargetAssemblyTask">
    <GenerateTemporaryTargetAssembly
      AssemblyName="WPFMSBuildSample"
      CompileTargetName="CoreCompile"
      CompileTypeName="Compile"
      CurrentProject="FullBuild.proj"
      GeneratedCodeFiles="obj\debug\app.g.cs;obj\debug\Page1.g.cs;obj\debug\Page2.g.cs"
      ReferencePath="c:\windows\Microsoft.net\Framework\v2.0.50727\System.dll;C:\Program Files\Reference
Assemblies\Microsoft\WinFx\v3.0\PresentationCore.dll;C:\Program Files\Reference
Assemblies\Microsoft\WinFx\v3.0\PresentationFramework.dll;C:\Program Files\Reference
Assemblies\Microsoft\WinFx\v3.0\WindowsBase.dll"
      IntermediateOutputPath=".\obj\debug\"
      MSBuildBinPath="$(MSBuildBinPath)"
      ReferencePathTypeName="ReferencePath"/>
  </Target>
</Project>
```

See also

- WPF MSBuild reference
- Task reference
- MSBuild reference
- Task reference
- Build a WPF application (WPF)
- WPF XAML browser applications overview

GetWinFXPath task

2/21/2019 • 2 minutes to read • Edit Online

The GetWinFXPath task returns the directory of the current Microsoft .NET Framework runtime.

Task parameters

PARAMETER	DESCRIPTION
`WinFXPath`	Optional **String** output parameter. Specifies the real path to the .NET Framework runtime.
`WinFXNativePath`	Required **String** parameter. Specifies the path to the native .NET Framework runtime.
`WinFXWowPath`	Required **String** parameter. Specifies the path to the Microsoft .NET Framework assemblies in the 32-bit **Windows on Windows** module on 64-bit systems.

Remarks

If the GetWinFXPath task is executing on a 64-bit processor, the **WinFXPath** parameter is set to the path that is stored in the **WinFXWowPath** parameter; otherwise, the **WinFXPath** parameter is set to the path that is stored in the **WinFXNativePath** parameter.

Example

The following example shows how to use the **GetWinFXPath** task to detect the native path to the .NET Framework runtime.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <UsingTask
    TaskName="Microsoft.Build.Tasks.Windows.GetWinFXPath"
    AssemblyFile="C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0\PresentationBuildTasks.dll"
/>
  <Target Name="GetWinFXPathTask">
    <GetWinFXPath
      WinFXNativePath="c:\WinFXNative"
      WinFXWowPath="c:\WinFXWowNative" />
  </Target>
  <Import Project="$(MSBuildBinPath)\Microsoft.WinFX.targets" />
</Project>
```

See also

- WPF MSBuild reference
- Task reference
- MSBuild reference

- Task reference
- Build a WPF application (WPF)

MarkupCompilePass1 task

2/21/2019 • 5 minutes to read • Edit Online

The MarkupCompilePass1 task converts non-localizable Extensible Application Markup Language (XAML) project files to compiled binary format.

Task parameters

PARAMETER	DESCRIPTION
`AllGeneratedFiles`	Optional **ITaskItem[]** output parameter. Contains a complete list of files that are generated by the MarkupCompilePass1 task.
`AlwaysCompileMarkupFilesInSeparateDomain`	Optional **Boolean** parameter. Specifies whether to run the task in a separate AppDomain. If this parameter returns **false**, the task runs in the same AppDomain as Microsoft build engine (MSBuild) and it runs faster. If the parameter returns **true**, the task runs in a second AppDomain that is isolated from MSBuild and runs slower.
`ApplicationMarkup`	Optional **ITaskItem[]** parameter. Specifies the name of the application definition XAML file.
`AssembliesGeneratedDuringBuild`	Optional **String[]** parameter. Specifies references to assemblies that change during the build process. For example, a Visual Studio solution may contain one project that references the compiled output of another project. In this case, the compiled output of the second project can be added to the **AssembliesGeneratedDuringBuild** parameter. Note: The **AssembliesGeneratedDuringBuild** parameter must contain references to the complete set of assemblies that are generated by a build solution.
`AssemblyName`	Required **string** parameter. Specifies the short name of the assembly that is generated for a project. For example, if a project is generating a Windows executable whose name is *WinExeAssembly.exe*, the **AssemblyName** parameter has a value of **WinExeAssembly**.
`AssemblyPublicKeyToken`	Optional **String** parameter. Specifies the public key token for the assembly.
`AssemblyVersion`	Optional **String** parameter. Specifies the version number of the assembly.

PARAMETER	DESCRIPTION
`ContentFiles`	Optional **ITaskItem[]** parameter. Specifies the list of loose content files.
`DefineConstants`	Optional **String** parameter. Specifies that the current value of **DefineConstants**, is kept. which affects target assembly generation; if this parameter is changed, the public API in the target assembly may be changed and the compilation of XAML files that reference local types may be affected.
`ExtraBuildControlFiles`	Optional **ITaskItem[]** parameter. Specifies a list of files that control whether a rebuild is triggered when the MarkupCompilePass1 task reruns; a rebuild is triggered if one of these files changes.
`GeneratedBamlFiles`	Optional **ITaskItem[]** output parameter. Contains the list of generated files in XAML binary format.
`GeneratedCodeFiles`	Optional **ITaskItem[]** output parameter. Contains the list of generated managed code files.
`GeneratedLocalizationFiles`	Optional **ITaskItem[]** output parameter. Contains the list of localization files that were generated for each localizable XAML file.
`HostInBrowser`	Optional **String** parameter. Specifies whether the generated assembly is a XAML browser application (XBAP). The valid options are **true** and **false**. If **true**, code is generated to support browser hosting.
`KnownReferencePaths`	Optional **String[]** parameter. Specifies references to assemblies that do not change during the build process. Includes assemblies that are located in the global assembly cache (GAC), in a Microsoft .NET Framework installation directory, and so on.
`Language`	Required **String** parameter. Specifies the managed language that the compiler supports. The valid options are **C#**, **VB**, **JScript**, and **C++**.

PARAMETER	DESCRIPTION
`LanguageSourceExtension`	Optional **String** parameter. Specifies the extension that is appended to the extension of the generated managed code file: `<Filename>.g<LanguageSourceExtension>` If the **LanguageSourceExtension** parameter is not set with a specific value, the default source file name extension for a language is used: *.vb* for Microsoft Visual Basic, *.csharp* for C#.
`LocalizationDirectivesToLocFile`	Optional **String** parameter. Specifies how to generate localization information for each source XAML file. The valid options are **None**, **CommentsOnly**, and **All**.
`OutputPath`	Required **String** parameter. Specifies the directory in which the generated managed code files and XAML binary format files are generated.
`OutputType`	Required **String** parameter. Specifies the type of assembly that is generated by a project. The valid options are **winexe**, **exe**, **library**, and **netmodule**.
`PageMarkup`	Optional **ITaskItem[]** parameter. Specifies a list of XAML files to process.
`References`	Optional **ITaskItem[]** parameter. Specifies the list of references from files to assemblies that contain the types that are used in the XAML files.
`RequirePass2ForMainAssembly`	Optional **Boolean** output parameter. Indicates whether the project contains non-localizable XAML files that reference local types that are embedded into the main assembly.
`RequirePass2ForSatelliteAssembly`	Optional **Boolean** output parameter. Indicates whether the project contains localizable XAML files that reference local types that are embedded in the main assembly.
`RootNamespace`	Optional **String** parameter. Specifies the root namespace for classes that are inside the project. **RootNamespace** is also used as the default namespace of a generated managed code file when the corresponding XAML file does not include the `x:Class` attribute.

PARAMETER	DESCRIPTION
`SourceCodeFiles`	Optional **ITaskItem[]** parameter. Specifies the list of code files for the current project. The list does not include generated language-specific managed code files.
`UICulture`	Optional **String** parameter. Specifies the satellite assembly for the UI culture in which the generated XAML binary format files are embedded. If **UICulture** is not set, the generated XAML binary format files are embedded in the main assembly.
`XAMLDebuggingInformation`	Optional **Boolean** parameter. When **true**, diagnostic information is generated and included in the compiled XAML in order to aid debugging.

Remarks

The MarkupCompilePass1 task typically compiles XAML into binary format and generates code files. If a XAML file contains references to types that are defined in the same project, its compilation to binary format is deferred by **MarkupCompilePass1** to a second markup compilation pass (**MarkupCompilePass2**). Such files must have their compilation deferred because they must wait until the referenced locally-defined types are compiled. However, if a XAML file has an `x:Class` attribute, MarkupCompilePass1 generates the language-specific code file for it.

A XAML file is localizable if it contains elements that use the `x:Uid` attribute:

```
<Page x:Class="WPFMSBuildSample.Page1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Uid="Page1Uid"
    >
  ...
</Page>
```

A XAML file references a locally-defined type when it declares an XML namespace that uses the `clr-namespace` value to refer to a namespace in the current project:

```
<Page x:Class="WPFMSBuildSample.Page1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:localNamespace="clr-namespace:WPFMSBuildSample"
    >
    <Grid>
      <Grid.Resources>
        <localNameSpace:LocalType x:Key="localType" />
      </Grid.Resources>
      ...
    </Grid>
</Page>
```

If any XAML file is localizable, or references a locally-defined type, a second markup compilation pass is required, which requires running the GenerateTemporaryTargetAssembly and then the MarkupCompilePass2.

Example

The following example shows how to convert three *Page* XAML files to binary format files. *Page1* contains a reference to a type, `Class1`, which is in the root namespace of the project and therefore, is not converted to binary format files in this markup compile pass. Instead, the GenerateTemporaryTargetAssembly is executed and is followed by the MarkupCompilePass2.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <UsingTask
    TaskName="Microsoft.Build.Tasks.Windows.MarkupCompilePass1"
    AssemblyFile="C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0\PresentationBuildTasks.dll"
/>
  <Target Name="MarkupCompilePass1Task">
    <MarkupCompilePass1
      AssemblyName="WPFMSBuildSample"
      Language="C#"
      OutputType="WinExe"
      OutputPath="obj\Debug\"
      ApplicationMarkup="App.xaml"
      PageMarkup="Page1.xaml;Page2.xaml;Page3.xaml"
      SourceCodeFiles="Class1.cs"
      References="c:\windows\Microsoft.net\Framework\v2.0.50727\System.dll;C:\Program Files\Reference
Assemblies\Microsoft\WinFx\v3.0\PresentationCore.dll;C:\Program Files\Reference
Assemblies\Microsoft\WinFx\v3.0\PresentationFramework.dll;C:\Program Files\Reference
Assemblies\Microsoft\WinFx\v3.0\WindowsBase.dll" />
  </Target>
</Project>
```

See also

- WPF MSBuild reference
- WPF MSBuild task reference
- MSBuild reference
- MSBuild task reference
- Build a WPF application (WPF)
- WPF XAML browser applications overview

MarkupCompilePass2 task

2/21/2019 • 2 minutes to read • Edit Online

The MarkupCompilePass2 task performs second-pass markup compilation on Extensible Application Markup Language (XAML) files that reference types in the same project.

Task parameters

PARAMETER	DESCRIPTION
`AlwaysCompileMarkupFilesInSeparateDomain`	Optional **Boolean** parameter. Specifies whether to run the task in a separate AppDomain. If this parameter returns **false**, the task runs in the same AppDomain as Microsoft build engine (MSBuild), and it runs faster. If the parameter returns **true**, the task runs in a second AppDomain that is isolated from MSBuild and runs slower.
`AssembliesGeneratedDuringBuild`	Optional **String[]** parameter. Specifies references to assemblies that change during the build process. For example, a Visual Studio solution may contain one project that references the compiled output of another project. In this case, the compiled output of the second project can be added to **AssembliesGeneratedDuringBuild**. Note: **AssembliesGeneratedDuringBuild** must contain references to the complete set of assemblies that are generated by a build solution.
`AssemblyName`	Required **String** parameter. Specifies the short name of the assembly that is generated for a project. For example, if a project is generating a Microsoft Windows executable whose name is *WinExeAssembly.exe*, the **AssemblyName** parameter has a value of **WinExeAssembly**.
`GeneratedBaml`	Optional **ITaskItem[]** output parameter. Contains the list of generated files in XAML binary format.
`KnownReferencePaths`	Optional **String[]** parameter. Specifies references to assemblies that are never changed during the build process. Includes assemblies that are located in the global assembly cache (GAC), in a Microsoft .NET Framework installation directory, and so on.
`Language`	Required **String** parameter. Specifies the managed language that the compiler supports. The valid options are **C#**, **VB**, **JScript**, and **C++**.

PARAMETER	DESCRIPTION
`LocalizationDirectivesToLocFile`	Optional **String** parameter. Specifies how to generate localization information for each source XAML file. The valid options are **None**, **CommentsOnly**, and **All**.
`OutputPath`	Required **String** parameter. Specifies the directory in which the generated XAML binary format files are generated.
`OutputType`	Required **String** parameter. Specifies the type of assembly that is generated by a project. The valid options are **winexe**, **exe**, **library**, and **netmodule**.
`References`	Optional **ITaskItem[]** parameter. Specifies the list of references from files to assemblies that contain the types that are used in the XAML files. One reference is to the assembly that was generated by the GenerateTemporaryTargetAssembly task, which must be run before the MarkupCompilePass2 task.
`RootNamespace`	Optional **String** parameter. Specifies the root namespace for classes that are inside the project. **RootNamespace** is also used as the default namespace of a generated managed code file when the corresponding XAML file does not include the `x:Class` attribute.
`XAMLDebuggingInformation`	Optional **Boolean** parameter. When **true**, diagnostic information is generated and included in the compiled XAML in order to aid debugging.

Remarks

Before you run **MarkupCompilePass2**, you must generate a temporary assembly that contains the types that are used by the XAML files whose markup compilation pass were deferred. You generate the temporary assembly by running the **GenerateTemporaryTargetAssembly** task.

A reference to the generated temporary assembly is provided to MarkupCompilePass2 when it runs, allowing the XAML files whose compilation was deferred in the first markup compilation pass to now be compiled to binary format.

Example

The following example shows how to use the MarkupCompilePass2 task to perform a second pass compilation.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <UsingTask
    TaskName="Microsoft.Build.Tasks.Windows.MarkupCompilePass2"
    AssemblyFile="C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0\PresentationBuildTasks.dll"
/>
  <Target Name="MarkupCompilePass2Task">
    <MarkupCompilePass2
      AssemblyName="WPFMSBuildSample"
      Language="C#"
      OutputType="WinExe"
      OutputPath="obj\Debug\"

References=".\obj\debug\WPFMSBuildSample.exe;c:\windows\Microsoft.net\Framework\v2.0.50727\System.dll;C:\Progr
am Files\Reference Assemblies\Microsoft\WinFx\v3.0\PresentationCore.dll;C:\Program Files\Reference
Assemblies\Microsoft\WinFx\v3.0\PresentationFramework.dll;C:\Program Files\Reference
Assemblies\Microsoft\WinFx\v3.0\WindowsBase.dll" />
  </Target>
</Project>
```

See also

- WPF MSBuild reference
- WPF MSBuild task reference
- MSBuild reference
- MSBuild task reference
- Build a WPF application (WPF)
- WPF XAML browser applications overview

MergeLocalizationDirectives task

2/21/2019 • 2 minutes to read • Edit Online

The MergeLocalizationDirectives task merges the localization attributes and comments of one or more XAML binary format files into a single file for the whole assembly.

Task parameters

PARAMETER	DESCRIPTION
`GeneratedLocalizationFiles`	Required **ITaskItem[]** parameter. Specifies the list of localization directives files for individual files in XAML binary format.
`OutputFile`	Required **String** output parameter. Specifies the output path of the compiled localization-directives assembly.

Remarks

You can add localization attributes and comments to Extensible Application Markup Language (XAML) content. With Windows Presentation Foundation (WPF) localization support, you can strip out localization attributes and comments, and put them in a *.loc* file that is separate from the generated assembly. You can do this by using the **LocalizationPropertyStorage** attribute. For more information about localization attributes and comments, and **LocalizationPropertyStorage**, see Localization attributes and comments.

Example

The following example merges the localization comments of several XAML binary format files into a single *.loc* file.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <UsingTask
    TaskName="Microsoft.Build.Tasks.Windows.MergeLocalizationDirectives"
    AssemblyFile="C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0\PresentationBuildTasks.dll"
/>
  <Target Name="MergeLocalizationDirectivesTask">
    <MergeLocalizationDirectives
      GeneratedLocalizationFiles="obj\debug\page1.loc;obj\debug\page2.loc;obj\debug\page3.loc"
      OutputFile="obj\debug\WPFMSBuildSample.loc" />
  </Target>
</Project>
```

See also

- WPF MSBuild reference
- WPF MSBuild task reference
- MSBuild reference
- MSBuild task reference
- Build a WPF application (WPF)

ResourcesGenerator task

2/21/2019 • 2 minutes to read • Edit Online

The ResourcesGenerator task embeds one or more resources (*.jpg*, *.ico*, *.bmp*, XAML in binary format, and other extension types) into a *.resources* file.

Task parameters

PARAMETER	DESCRIPTION
`OutputPath`	Required **String** parameter. Specifies the path of the output directory. If the path isn't an absolute path, it's treated as a path that is relative to the root project directory.
`OutputResourcesFile`	Required **ITaskItem[]** output parameter. Specifies the path and name of the generated *.resources* file. If the path isn't an absolute path, the *.resources* file is generated relative to the root project directory.
`ResourcesFiles`	Required **ITaskItem[]** parameter. Specifies one or more resources to embed in the generated *.resources* file.

Example

The following example generates a *.resources* file with a single *.bmp* resource. The *.bmp* resource is generated to a directory that is relative to the project root directory.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <UsingTask
    TaskName="Microsoft.Build.Tasks.Windows.ResourcesGenerator"
    AssemblyFile="C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0\PresentationBuildTasks.dll"
/>
  <Target Name="ResourcesGeneratorTask">
    <ResourcesGenerator
      ResourceFiles="Resource1.bmp"
      OutputPath="myresources"
      OutputResourcesFile="myresources\my.resources" />
  </Target>
</Project>
```

See also

- WPF MSBuild reference
- Task reference
- MSBuild reference
- Task reference
- Build a WPF application (WPF)

UidManager task

2/21/2019 • 2 minutes to read • Edit Online

The UidManager task checks, updates, or removes unique identifiers (UIDs), in order to localize all Extensible Application Markup Language (XAML) elements that are included in the source XAML files.

Task parameters

PARAMETER	DESCRIPTION
`IntermediateDirectory`	Optional **String** parameter. Specifies the directory that is used to back up the source XAML files that are specified by the **MarkupFiles** parameter.
`MarkupFiles`	Required **ITaskItem[]** parameter. Specifies the source XAML files to include for UID checking, updating, or removing.
`Task`	Required **String** parameter. Specifies the UID management task that you want to perform. Valid options are **Check**, **Update**, or **Remove**.

Example

The following example uses the UidManager task to check that the specified source XAML files contain XAML elements that have appropriate UIDs.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <UsingTask
    TaskName="Microsoft.Build.Tasks.Windows.UidManager"
    AssemblyFile="C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0\PresentationBuildTasks.dll"
/>
  <Target Name="UidManagerTask">
    <UidManager
      Task="Check"
      MarkupFiles="Page1.xaml;Page2.xaml"
      IntermediateDirectory="c:\UidManagerIntermediateDirectory" />
  </Target>
</Project>
```

See also

- WPF MSBuild reference
- Task reference
- MSBuild reference
- Task reference
- Build a WPF application (WPF)
- How to: Localize an application

UpdateManifestForBrowserApplication task

2/21/2019 • 2 minutes to read • Edit Online

The UpdateManifestForBrowserApplication task is run to add the **<hostInBrowser />** element to the application manifest (<*projectname*>.*exe.manifest*) when a XAML browser application (XBAP) project is built.

Task parameters

PARAMETER	DESCRIPTION
`ApplicationManifest`	Required **ITaskItem[]** parameter. Specifies the path and name of the application manifest file that you want to add the `<hostInBrowser />` element to.
`HostInBrowser`	Required **Boolean** parameter. Specifies whether to modify the application manifest to include the **<hostInBrowser />** element. If **true**, a new **<hostInBrowser />** element is included in the **<entryPoint />** element. Element inclusion is cumulative: if a **<hostInBrowser />** element already exists, it isn't removed or overwritten. Instead, an additional **<hostInBrowser />** element is created. If **false**, the application manifest isn't modified.

Remarks

XBAPs are run by using ClickOnce deployment, so they must be published with supporting deployment and application manifests. Microsoft build engine (MSBuild) uses the GenerateApplicationManifest task to generate an application manifest.

Then, to configure an application to be hosted from a browser, an additional **<hostInBrowser />** element must be added to the application manifest, as shown in the following example:

```
<!--MyXBAPApplication.exe.manifest-->
<?xml version="1.0" encoding="utf-8"?>
<asmv1:assembly ... >
    <asmv1:assemblyIdentity ... />
    <application />
    <entryPoint>
      ...
      <hostInBrowser xmlns="urn:schemas-microsoft-com:asm.v3" />
    </entryPoint>
  ...
/>
```

The UpdateManifestForBrowserApplication task is run when an XBAP project is built in order to add the `<hostInBrowser />` element.

Example

The following example shows how to make sure that the `<hostInBrowser />` element is included in an application

manifest file.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <UsingTask
    TaskName="Microsoft.Build.Tasks.Windows.UpdateManifestForBrowserApplication"
    AssemblyFile="C:\Program Files\Reference Assemblies\Microsoft\Framework\v3.0\PresentationBuildTasks.dll"
/>
  <Target Name="UpdateManifestForBrowserApplicationTask">
    <UpdateManifestForBrowserApplication
      ApplicationManifest="MyXBAPApplication.exe.manifest"
      HostInBrowser="true" />
  </Target>
</Project>
```

See also

- WPF MSBuild reference
- Task reference
- MSBuild reference
- Task reference
- Build a WPF application (WPF)
- WPF XAML browser applications overview

Special characters to escape

9/27/2019 • 2 minutes to read • Edit Online

Special characters must be escaped only if they have special meaning in the context in which they are being used. For example, the asterisk (*) is a special character only in the "Include" and "Exclude" attributes of an item definition, or in a call to CreateItem. In all other cases, the asterisk is treated as a literal asterisk. While you do not need to escape asterisks everywhere in project files, doing so does no harm.

Use the notation %<xx> in place of the special character, where <xx> represents the hexadecimal value of the ASCII character. For example, to use an asterisk (*) as a literal character, use the value `%2A`.

The full list of special characters to escape follows:

CHARACTER	DESCRIPTION
%	Percent sign, used to reference metadata.
$	Dollar sign, used to reference properties.
@	At sign, used to reference item lists.
(	Open parenthesis, used in lists.
)	Close parenthesis, used in lists.
;	Semicolon, a list separator.
?	Question mark, a wildcard character when describing a file spec in an item's Include/Exclude section.
*	Asterisk, a wildcard character when describing a file spec in an item's Include/Exclude section.

NOTE

In some scenarios, you may need to escape double quote (") characters, such as when using within an `Exec` task.

See also

- How to: Escape special characters in MSBuild
- MSBuild reference

Use the MSBuild API

2/21/2019 • 2 minutes to read • Edit Online

MSBuild provides a public API surface so that your program can perform builds and inspect projects.

Documentation for the MSBuild API can be found at Microsoft.Build namespaces.

Update an existing application for MSBuild 15

10/24/2019 • 2 minutes to read • Edit Online

In versions of MSBuild prior to 15.0, MSBuild was loaded from the Global Assembly Cache (GAC) and MSBuild extensions were installed in the registry. This ensured all applications used the same version of MSBuild and had access to the same Toolsets, but prevented side-by-side installations of different versions of Visual Studio.

To support faster, smaller, and side-by-side installation, Visual Studio 2017 and later versions no longer place MSBuild in the GAC or modifies the registry. Unfortunately, this means that applications that wish to use the MSBuild API to evaluate or build projects can't implicitly rely on the Visual Studio installation.

Use MSBuild from Visual Studio

To ensure that programmatic builds from your application match builds done within Visual Studio or *MSBuild.exe*, load MSBuild assemblies from Visual Studio and use the SDKs available within Visual Studio. The Microsoft.Build.Locator NuGet package streamlines this process.

Use Microsoft.Build.Locator

If you redistribute *Microsoft.Build.Locator.dll* with your application, you won't need to distribute other MSBuild assemblies.

Updating a project to use MSBuild 15 and the locator API requires a few changes in your project, described below. To see an example of the changes required to update a project, see the commits made to an example project in the MSBuildLocator repository.

Change MSBuild references

To make sure that MSBuild loads from a central location, you must not distribute its assemblies with your application.

The mechanism for changing your project to avoid loading MSBuild from a central location depends on how you reference MSBuild.

Use NuGet packages (preferred)

These instructions assume that you're using PackageReference-style NuGet references.

Change your project file(s) to reference MSBuild assemblies from their NuGet packages. Specify `ExcludeAssets=runtime` to tell NuGet that the assemblies are needed only at build time, and shouldn't be copied to the output directory.

The major and minor version of the MSBuild packages must be less than or equal to the minimum version of Visual Studio you wish to support. For example, if you wish to support Visual Studio 2017 and later versions, reference package version `15.1.548`.

For example, you can use this XML:

```
<ItemGroup>
  <PackageReference Include="Microsoft.Build" Version="15.1.548" ExcludeAssets="runtime" />
  <PackageReference Include="Microsoft.Build.Utilities.Core" Version="15.1.548" ExcludeAssets="runtime" />
</ItemGroup>
```

Use extension assemblies

If you can't use NuGet packages, you can reference MSBuild assemblies that are distributed with Visual Studio. If

you reference MSBuild directly, ensure that it won't be copied to your output directory by setting `Copy Local` to `False`. In the project file, this setting will look like the following code:

```
    <Reference Include="Microsoft.Build, Version=15.1.0.0, Culture=neutral, PublicKeyToken=b03f5f7f11d50a3a,
processorArchitecture=MSIL">
      <Private>False</Private>
    </Reference>
```

Binding redirects

Reference the Microsoft.Build.Locator package to ensure that your application automatically uses the required binding redirects to version 15.1.0.0. Binding redirects to this version support both MSBuild 15 and MSBuild 16.

Ensure output is clean

Build your project and inspect the output directory to make sure that it doesn't contain any *Microsoft.Build.*.dll* assemblies other than *Microsoft.Build.Locator.dll*, added in the next step.

Add package reference for Microsoft.Build.Locator

Add a NuGet package reference for Microsoft.Build.Locator.

```
    <PackageReference Include="Microsoft.Build.Locator">
      <Version>1.1.2</Version>
    </PackageReference>
```

Do not specify `ExcludeAssets=runtime` for the Microsoft.Build.Locator package.

Register instance before calling MSBuild

Add a call to the Locator API before calling any method that uses MSBuild.

The simplest way to add the call to the Locator API is to add a call to

```
MSBuildLocator.RegisterDefaults();
```

in your application startup code.

If you would like finer-grained control over the loading of MSBuild, you can select a result of `MSBuildLocator.QueryVisualStudioInstances()` to pass to `MSBuildLocator.RegisterInstance()` manually, but this is generally not needed.

MSBuild glossary

8/9/2019 • 7 minutes to read • Edit Online

These terms are used to describe the Microsoft Build Engine (MSBuild) and its components.

Glossary

AssemblyFoldersEx
A registry location where third party vendors store paths for each version of the framework that they support where design time resolution can look to find reference assemblies.

batching
Batching means dividing items into different categories known as *batches*, based on item metadata, and then running a target or task one time by using each batch. Batching is the MSBuild equivalent of the for--loop construct. For more information, see Batching.

build-scope
Build-scope describes an MSBuild object, for example, a global property, that is potentially visible to a project and to any child projects that are created in a multi-project build.

child project
See *project, child*.

condition
Many MSBuild elements can be defined conditionally; that is, the `Condition` attribute appears in the element. The contents of conditional elements are ignored unless the condition evaluates to `true`. For more information, see Conditions.

definition, item
See *item definition*.

emit item
During the execution phase of a build, items can be created or modified by tasks that have child `Output` elements that have the `ItemName` attribute. The task is said to "emit" the new items.

emit property
During the execution phase of a build, properties can be created or modified by tasks that have child `Output` elements that have the `PropertyName` attribute. The task is said to "emit" the new property.

evaluation phase
Evaluation is the first phase of a project build. All properties and items are evaluated in the order in which they appear in the project. Imported projects are evaluated as they are encountered in the project. Targets and tasks are not run until the execution phase, and any properties or items they would declare or emit are ignored during evaluation.

execution phase
Execution is the second phase of a project build. Selected targets are built and tasks are run. Properties and items can be created or modified compared to their evaluation values.

function, property
See *property function*.

function, item

See item function.

item
Items are inputs into the build system, and are grouped into item types based on their element names. Items typically represent files. Because items are named by the item type they belong to, the terms *item* and *item value* can be used interchangeably. For more information, see Items.

item definition
Item definition groups contain item definitions that add default metadata to any item type. Like well-known metadata, the default metadata is associated with all items of the specified item type. Default metadata can be explicitly overridden in an item definition. For more information, see Item definitions.

item function
Item functions get information about the items in the project. These functions simplify getting Distinct() items and are faster than looping through the items. There are functions to manipulate item paths and strings. For more information, see Item functions.

item metadata
See *metadata, item*.

item type
Item types are named lists of items that can be used as parameters for tasks. The tasks use the item values to perform the steps of the build process. For more information, see Items.

metadata, item
Item metadata is a collection of name-value pairs that is associated with an item. Metadata provides descriptive information for the item and is optional, except for well-known metadata. For more information, see Items.

metadata, well-known
Well-known metadata is read-only item metadata that is initialized by using a predefined value. Well-known metadata provides descriptive information for an item that references a file. For example, the value of the well-known metadata named `FullPath` is the full path of the referenced file. For more information, see Items.

multitargeting
The ability for an application or assembly project to target many different CLR's and frameworks from MSBuild and from Visual Studio.

profile
A subset of the full framework. This is used to minimize the amount that needs to be downloaded to a machine.

project file
A project file contains the MSBuild script that controls the build. Project files typically have a file extension that ends with *proj*, such as *.csproj* or *.vbproj*. Project files may import property files and target files.

property
A property is a key-value pair that is used to control the build process. For more information, see MSBuild properties.

property, environment
An environment property is a property that is automatically initialized to the value of a system environment variable that has the same name. For more information, see MSBuild properties.

property file
A property file is a project file that contains mostly property groups and item groups that guide the build. By convention, It has the file extension *.props*. Property files are typically imported at the beginning of associated project files.

property, function

A property function is a system property or method that can be used to evaluate MSBuild scripts. Property methods can be used to read the system time, compare strings, match regular expressions, and perform other actions. For more information, see Property functions.

property function, nested
Property functions may be combined to form more complex functions. For example,

```
$([MSBuild]::BitwiseAnd(32, $([System.IO.File]::GetAttributes(tempFile))))
```

For more information, see Property functions.

property, global
A global property is a key-value pair that is used to control the build process. Global properties are set at a command prompt, or by using the `Properties` attribute of an MSBuild task, and cannot be modified during the evaluation phase of a build. For more information, see MSBuild properties.

property, local
A local property is a key-value pair that is used to control the build process. This term is only used to distinguish a property that is not a global property.

property, registry
A registry property has a value that is set by using a special syntax that reads the value of a system registry subkey. For more information, see MSBuild properties.

property, reserved
A reserved property is a key-value pair that is used to control the build process. Reserved properties are automatically initialized to predefined values. For more information, see MSBuild properties.

project-scope
Project-scope describes an MSBuild object, for example, a local property, that is visible only in the containing project file and to any projects that it imports.

project, child
A child project is created by the MSBuild task during a project build. This new project is a child of the project that contains or imports the target that contains the MSBuild task. The child project inherits the global properties of the parent project, unless they are modified by the `Properties` attribute.

redist list
Redistribution list: the list of assemblies that correspond to a given framework.

reference assembly
An assembly that is used during design time to create an application. A reference assembly can have the actual code and private interfaces removed from it, leaving only the metadata and public interfaces.

registry property
See *property, registry*.

target
A target groups tasks together in a particular order and exposes sections of the project file as entry points into the build process. For more information, see Targets.

target, building
See target, running.

target, evaluating
Because of incremental compilation, targets must be analyzed for potential changes to properties and items. Even if the target is skipped, these changes must be made. Evaluating a target means performing this analysis and making these changes. For more information, see Incremental builds.

target, executing
Executing a target means evaluating it and executing all tasks that have no conditions, or whose conditions evaluate to true. During incremental compilation, targets may be skipped or executed, but they are always evaluated. For more information, see target, evaluating.

target, running
A target that has a condition that evaluates to false is not run, that is, has no effect on the build. Targets that run are either executed or skipped. In either case, the target is evaluated. For more information, see target, evaluating.

target, skipping
If incremental compilation determines that all output files are up-to-date, then the target is skipped, that is, the target is evaluated, but the tasks within the target are not executed. For more information, see target, evaluating.

target framework moniker
A name that describes the framework (such as .NETFramework, Silverlight, etc.), the version, and the profile (such as Client, Server, etc.) that you wish to target.

targeting pack
The list of assemblies that are distributed with a given framework and the set of reference assemblies for that framework.

targets file
A targets file is a project file that contains mostly targets and tasks that guide the build. By convention, It has the file extension *.targets*. Target files are typically imported at the end of associated project files.

task
Tasks are units of executable code that MSBuild projects use to perform build operations. For example, a task might compile input files or run an external tool. For more information, see Tasks.

transform
A transform is a one-to-one conversion of one item collection to another. In addition to enabling a project to convert item collections, a transform enables a target to identify a direct mapping between its inputs and outputs. For more information, see Transforms.

well-known metadata
See *metadata, well-known*.

See also

- MSBuild