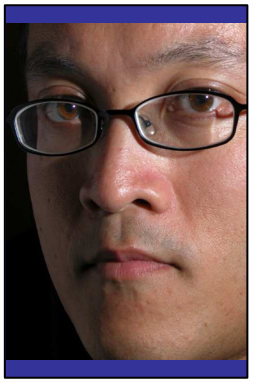
practical eye

FOR THE .NET GUY

Issue 2: January, 2004

Why do I do this?



It was 4:30 am on September 15, 1983. I was hunched over my desk in the basement of my parent's house in Scarborough, Canada. Another full day of Grade 10 beckoned at Agincourt Collegiate, yet I hadn't gone to bed yet. I *couldn't* go to bed yet.

I basked in the pale blue glow of my Commodore 64's 1701 monitor. I was doggedly inserting NOP statements into a critical piece of timing code. I was attempting to do something that had never been done before: asynchronously transmit two bits at a time between my 1541 floppy drive and my Commodore 64.

Moments later, I tested my creation. A wave of euphoria swept over me as I saw the results; I had just created *the* key piece of code that would boost the performance of Commodore's 1541 disk drive by over 500%!

I realized then and there that I write software because I can't *not* do it. It's a creative outlet for me, much like photography is. What's more, I feel compelled to share my creativity with others.

I have created this newsletter to share my latest research and ideas with you. In each issue, you will find an article that will help your software development team write better code, faster. To help ease the stress from long hours of coding, I also present you with a photograph from my portfolio.

I invite you to call me to discuss how together, we can help your team write better code, faster. You can reach me directly at 416-716-3718 or jlam@iunknown.com.

Cheers,

John Lam

Pipeline

By the time you read this, the **PDC** will be but a fond distant memory. Don't worry if you didn't make it there, however. If you have a group of developers in the Greater **Toronto** Area, and you want to get a **free** two-hour lunchtime briefing on **what was presented at the PDC**, please contact me ASAP and I'll make arrangements to come out to your company!

In the first issue of this newsletter, I alluded to a **key piece of technology** that will be used extensively by all developers working on the new platform. That new piece of technology was unveiled at the PDC, and is called **MSBuild**. This newsletter presents my **Practical Developers' Guide to MSBuild**. My goal is to get you productive using the new build system as quickly as possible.

This newsletter is generated using a **publishing framework** that I created using **open source** tools. Right now I'm working out the kinks in the system, and I'll be opening up the CVS project for others to collaborate on improving it in the near future. A future installment of this newsletter will discuss the details of my publishing framework.

A Practical Developers' Guide to MSBuild

I wrote this guide to get you writing your first MSBuild scripts within 30 minutes. It assumes that you are using the PDC build of Visual Studio .NET code-named "Whidbey".

When you build a solution using Visual Studio .NET "Whidbey", you are actually using MSBuild. MSBuild uses the information contained in the `.csproj` or `.vbproj` files that define a project in Visual Studio .NET.[1] One of the key goals of the MSBuild team is to allow developers to build MSBuild projects *without* VS.NET. This makes MSBuild more cost effective for development teams that create integration builds on a dedicated server since they no longer need to have a VS.NET license for that server.

When you examine those project files, you'll find that they bear little resemblance to the XML build files in other build systems such as [N]Ant.[2] The reason for the difference has to do with the project composition features of MSBuild, and the default project files[3] that ship with the product.

To understand *why* project composition is so important in MSBuild, we're going to spend some time re-creating a subset of the default project files, Along the

[1] In the Whidbey Alpha preview that was released at the PDC, MSBuild isn't used to build C++ or J# projects; only C# and VB.NET projects use MSBuild in those bits.

[2] When I say [N]Ant, I mean both Ant, the original build tool from the Apache Software Foundation (http://ant.apache.org), and NAnt, the .NET clone of Ant (http://nant.sourceforge.net).

[3] You can find these project files in your .NET system directory: \Windows\Microsoft.Net\Framework\V1.2.30703*.targets.

way, I'll introduce most of the features of MSBuild. By the time you finish reading this guide, you will understand most of the core concepts of MSBuild, and be able to understand the schema of the `.csproj` and `.vbproj` files.

Stand-alone MSBuild scripts

MSBuild scripts are just XML files, so they can be created using your favorite text editor. Let's begin our tour by writing a script that compiles this simple "Hello, World" application:

```
using System;

public class App
{
  public static void Main()
  {
    Console.WriteLine("Hello, World");
  }
}
```

Here's the MSBuild script that we will use to compile `app.cs`:

```
<Project DefaultTargets="Build">

  <Target Name="Build">

    <Task Name="Csc"
          Sources="App.cs"
          EmitDebugInformation="True"
          Optimize="False"
          DefineConstants="DEBUG"
          TargetType="Exe"/>

  </Target>
</Project>
```

By convention, all MSBuild files have a `.*proj` filename extension. Each MSBuild file defines a separate project. A project contains one or more targets, and a target contains one or more tasks.

Projects can generate one or more outputs. These outputs are represented by targets. In this simple project, we have a single target called `build` that creates an assembly called `app.exe`. To create `app.exe`, it uses the `csc` task to invoke the C# compiler to compile `app.cs` into `app.exe`.

In this example, I used three attributes of the `csc` task. The `EmitDebugInformation` attribute instructs the compiler to generate a program database (PDB) file. The `TargetType` attribute instructs the C# compiler to generate an executable assembly. The `Sources` attribute contains a list of files that will be consumed by the C# compiler. Although we are compiling only one file in this

case, if you wanted to compile multiple files you can provide a semicolon-delimited list of filenames, as in "`file1.cs; file2.cs; file3.cs`".

If you are familiar with the command-line switches for the C# compiler, you will notice that the attribute names are *not* the same as the switch names. The MSBuild philosophy is to err on the side of providing meaningful attribute names. For example, consider that a typical debug build requires setting *three* switches: `/debug+` and `/optimize-`, and `/d:DEBUG`. The `csc` task maps these three switches to the `EmitDebugInformation`, `Optimize`, and `DefineConstants` attributes.

To invoke MSBuild from the command line, all you need to do is type `msbuild`.[4] Here's the output from MSBuild:

```
Microsoft (R) .NET Build Engine version 1.2.30703.4
[Microsoft .Net Framework, Version 1.2.30703.4]
Copyright (C) Microsoft Corporation 2003. All rights
reserved.

Target "Build" in project "app.proj"
   Task "Csc"
      Csc.exe /define:DEBUG /debug+ /optimize-
/target:exe /out:App.exe "App.cs"
```

Properties and Items

In the previous example, I generated a debug build of `app.exe` from `app.cs`. However, all of the inputs to the `csc` task were hard-coded into the build script. In this step, I'll introduce the concepts of properties and items in MSBuild. Here's the updated `app.proj` file, rewritten to use properties and items:

```
<Project DefaultTargets="Build">

  <PropertyGroup>
    <Property DebugSymbols="True"/>
    <Property Optimize="False"/>
    <Property OutputType="exe"/>
  </PropertyGroup>

  <ItemGroup>
    <Item Type="Compile" Include="App.cs"/>
  </ItemGroup>

  <Target Name="Build">

    <Task Name="Csc"
          Sources="@(Compile)"
          EmitDebugInformation="$(DebugSymbols)"
          Optimize="$(Optimize)"
          DefineConstants="DEBUG"
          TargetType="$(OutputType)"/>

  </Target>

</Project>
```

[4] If you have several .proj files in the same directory, you must identify the .proj file that you want to build.

Notice that there is now a `PropertyGroup` element at the top of the file. Property groups are used as containers for individual properties.[5] Inside the `PropertyGroup` element, you will find a set of `Property` elements. Each `Property` element contains exactly one name-value pair.

Next, turn your attention to the revised Build target. Notice that the `csc` task is parameterized using property references. The properties that I defined within `PropertyGroup` are referred to as `$(DebugSymbols)`, `$(Optimize)`, and `$(OutputType)` within the `csc` task. When the task is executed, the property values are substituted at run-time.

MSBuild also has a concept of an `Item`. You can think of an `Item` as a collection of values.[6] `Items` are identified by a `Type` attribute, and their values are assigned using an `Include` attribute. The `Include` attribute can contain a path, a semi-colon delimited list of paths, or a wildcard.[7]

In this project, I've defined a new `Item` called `Compile`. The `Sources` attribute of the `csc` task refers to this `Item` as `@(Compile)`. At run-time, MSBuild generates the following command-line call to the C# compiler:

```
Csc.exe /debug+ /optimize- /target:exe
/out:App.exe "App.cs"
```

If there was more than one file in the `Compile` item, those files would be sent as arguments to the C# compiler.

[5] We will see how useful property groups are in practice when I introduce conditionals in the next step.

[6] This is unlike a property which has a single value.

[7] The wildcard syntax can be standard DOS syntax (e.g. *.cs) or it can be an Ant-style recursive syntax (e.g. **/*.cs which recursively refers to all files with a .cs extension in the current directory and all subdirectories under the current directory).

In this step, I've parameterized all of the inputs to the `csc` task using either properties or items. In the next step, I'll show you how you can assign different sets of properties to a task based on conditions that you can assign at runtime.

Multiple Configurations

Most software projects generate at least two configurations: a Debug build and a Release build. In this step, I will use MSBuild's conditional task evaluation feature to add support for Debug and Release builds to our sample project. Here's the updated `app.proj` file, re-written to use conditional task evaluation:

```
<Project DefaultTargets="Build">

  <PropertyGroup>
    <Property Configuration="Debug" Condition="
'$(Configuration)' == '' "/>
    <Property OutputType="exe"/>
  </PropertyGroup>

  <PropertyGroup Condition=" '$(Configuration)' ==
'Release' ">
    <Property DebugSymbols="False"/>
    <Property Optimize="True"/>
  </PropertyGroup>

  <PropertyGroup Condition=" '$(Configuration)' ==
'Debug' ">
    <Property DebugSymbols="True"/>
    <Property Optimize="False"/>
  </PropertyGroup>

  <ItemGroup>
    <Item Type="Compile" Include="App.cs"/>
  </ItemGroup>

  <Target Name="Build">

    <Task Name="Csc"
          Sources="@(Compile)"
          EmitDebugInformation="$(DebugSymbols)"
          DefineConstants="DEBUG"
          Optimize="$(Optimize)"
          TargetType="$(OutputType)"/>

  </Target>

</Project>
```

The first thing to notice is the presence of the three `PropertyGroup` elements in this file. The first `PropertyGroup` sets configuration-independent properties. Within this element, the first `Property` element sets the `Configuration` property to Debug if the user didn't define a `Configuration` property on the command line. To pass properties to an MSBuild project, you can use the `/p:PropertyName=Value` switch on the command line. The second `Property` element sets the `OutputType` property to `exe`, which instructs the C# compiler to emit a console mode application.

The second and third `PropertyGroups` set Release and Debug configuration properties respectively. You'll

notice that I'm using the `Condition` attribute to control whether the contents of that `PropertyGroup` are evaluated at run-time. While the `Condition` attribute is also supported by `Property` elements, evaluating the conditional in the `PropertyGroup` element results in a simpler and more readable project file.

This project represents the simplest type of conditional compilation; we simply change a few switches on the C# compiler. In the next step, I'll add a few real-world improvements to our build configuration: the ability to specify the filename of the output assembly as well as the build's output directory.

Conditional Properties

Filenames in Windows are governed by some well-known naming conventions. An `.exe` represents a Win32 executable module, a `.dll` represents a Win32 library module, and a `.netmodule` represents a .NET module. In .NET, there is also the well-known naming convention that says that an assembly's name is the same as its filename minus its filename extension.

Based on these naming conventions, the minimum amount of information that we need to provide in our project is the assembly's name and type. This is done using the `AssemblyName` and `OutputType` properties in the updated `app.proj` project file:

```
<Project DefaultTargets="Build">

  <PropertyGroup>
    <Property Configuration="Debug"
              Condition=" '$(Configuration)' == '' "/>
    <Property AssemblyName="$(MSBuildProjectName)"/>
    <Property OutputPath=".\bin\$(Configuration)\"/>
    <Property OutputType="exe"/>
  </PropertyGroup>

  <PropertyGroup Condition=" '$(Configuration)' ==
'Release' ">
    <Property DebugSymbols="False"/>
    <Property Optimize="True"/>
  </PropertyGroup>

  <PropertyGroup Condition=" '$(Configuration)' ==
'Debug' ">
    <Property DebugSymbols="True"/>
    <Property Optimize="False"/>
  </PropertyGroup>

  <PropertyGroup>
    <Property TargetExt="exe"
              Condition="'$(OutputType)'=='exe'"/>
    <Property TargetExt="exe"
              Condition="'$(OutputType)'=='winexe'"/>
    <Property TargetExt="dll"
              Condition="'$(OutputType)'=='library'"/>
    <Property TargetExt="netmodule"
              Condition="'$(OutputType)'=='module'"/>
    <Property
OutputAssembly="$(OutputPath)$(AssemblyName).$(TargetExt
)"/>
  </PropertyGroup>
```

```
  <Item Type="DefinedSymbols"
        Include="DEBUG"
        Condition=" '$(Configuration)' == 'Debug' "/>

  <ItemGroup>
    <Item Type="Compile" Include="App.cs"/>
  </ItemGroup>

  <Target Name="CreateOutputDirectories"
          Condition="!Exists('$(OutputPath)')">

    <Task Name="MakeDir"
          Directories="$(OutputPath)"/>

  </Target>

  <Target Name="Build"
          DependsOnTargets="CreateOutputDirectories">

    <Task Name="Csc"
          Sources="@(Compile)"
          EmitDebugInformation="$(DebugSymbols)"
          DefineConstants="@(DefinedSymbols)"
          Optimize="$(Optimize)"
          TargetType="$(OutputType)"
          OutputAssembly="$(OutputAssembly)"/>

  </Target>

</Project>
```

The `OutputType` property can take on one of four values: exe, winexe, library, and module. These values represent the four different assembly types that can be generated by a .NET compiler. We can use this property to derive the filename extension of the generated assembly. The code in the fourth `PropertyGroup` element uses the `Condition` attribute of each `Property` element to assign the correct filename extension to the `TargetExt` property.

Next, let's turn our attention to the build's output directory. In Visual Studio .NET C# projects, the default output directory for a debug build is the project's `bin\Debug` directory, and the default output of a release build is the project's `bin\Release` directory. In this project file, I use the `OutputPath` property to define the location of the output directory. Notice that this property is defined differently in the two `PropertyGroup` elements that conditionally define the properties for the Debug and Release configurations.

In this project, I use the `OutputAssembly` property to tell the compiler where to put the generated assembly. This property is defined in the fourth `PropertyGroup` element; notice how I generate the `OutputAssembly` property by combining the `OutputPath`, `AssemblyName`, and `TargetExt` properties. In the Build target, I assign

this property to the `csc` task's `OutputAssembly` attribute.

Since we now specify the output directory, we need to create it if it doesn't exist already. This is what the `CreateOutputDirectories` target does in this project. It uses the `MakeDir` task to create the directory specified by the `OutputPath` property.[8] The `Build` target is still the default target of this project. Notice how it now sports a `DependsOnTargets` attribute to indicate that it requires that `CreateOutputDirectories` to run successfully before it can run.

You can see that the targets in this project are quite generic. Both the `CreateOutputDirectories` and the `Build` targets are completely parameterized. In the next step, we'll see how we can use the `Import` feature of MSBuild to take advantage of our generic targets, and greatly simplify the project files that you have to write.

Partitioning Projects

MSBuild's Import feature lets you partition a project file into two or more independent files. In this step, I've introduced no new features. Instead, I've refactored the project file from the previous step into three separate files. First, there's the new `app.proj` file, which now contains information specific only to this project:

```
<Project DefaultTargets="Build">

  <PropertyGroup>
    <Property AssemblyName="App"/>
    <Property OutputType="exe"/>
  </PropertyGroup>

  <ItemGroup>
    <Item Type="Compile" Include="App.cs"/>
  </ItemGroup>

  <Import Project="common.properties"/>
  <Import Project="common.targets"/>

</Project>
```

You'll notice right away that this file contains the bare minimum of information required to build this project. The `Compile` Item includes the only source code file in this

[8] Ideally, I would delete the directory first before recreating it during a full build. However, the Alpha version of MSBuild that I'm using for this article doesn't include a Delete task. This task will be present in the final release of the product.

project: App.cs. The assembly name that is generated is App, and that it s going to be a console-mode executable. The key to the simplification of this project are the two Import elements in this file. They import two additional project files: common.properties and common.targets.[9]

The common.properties file contains all of the default property declarations that we had in our previous project file:

```
<Project>

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

  <PropertyGroup Condition=" '$(Configuration)' ==
'Release' ">
    <Property DebugSymbols="False"/>
    <Property Optimize="True"/>
    <Property OutputPath=".\bin\Release\"/>
  </PropertyGroup>

  <PropertyGroup Condition=" '$(Configuration)' ==
'Debug' ">
    <Property DebugSymbols="True"/>
    <Property Optimize="False"/>
    <Property OutputPath=".\bin\Debug\"/>
  </PropertyGroup>

  <PropertyGroup>
    <Property TargetExt="exe"
Condition="'$(OutputType)'=='exe'"/>
    <Property TargetExt="exe"
Condition="'$(OutputType)'=='winexe'"/>
    <Property TargetExt="dll"
Condition="'$(OutputType)'=='library'"/>
    <Property TargetExt="netmodule"
Condition="'$(OutputType)'=='module'"/>
    <Property
OutputAssembly="$(OutputPath)$(AssemblyName).$(TargetExt
)"/>
  </PropertyGroup>

</Project>
```

This file contains properties that are common to all projects.

The common.targets file contains all of the targets from the previous project file:

```
<Project>

  <Target Name="CreateOutputDirectories">

    <Task Name="MakeDir"
          Directories="$(OutputPath)"/>

  </Target>

  <Target Name="Build"
DependsOnTargets="CreateOutputDirectories">

    <Task Name="Csc"
          Sources="@(Compile)"
          EmitDebugInformation="$(DebugSymbols)"
          Optimize="$(Optimize)"
          TargetType="$(OutputType)"
          OutputAssembly="$(OutputAssembly)"/>

  </Target>
```

[9] While reviewing this newsletter, Alex Kipman the MSBuild program manager at Microsoft, wondered why I had a properties file *and* a targets file. I replied that I like separating code from data, and that I felt that most folks would prefer just modifying the properties file without having to be exposed to the complexities of the targets file.

```
</Project>
```

In this and all the previous steps, you did a full build every time you ran your build script. Both targets executed regardless of whether the underlying files required recompilation. In the next step, I will introduce a way to define whether a target should be executed: through the target's `Inputs` and `Outputs` attributes.

Incremental Builds

Up until this step, all our builds have all been *full builds*[10]. In this step, I'm going to implement a minimal build by modify the `common.targets` file from the previous step:

```
<Project>

  <Import Project="common.properties"/>

  <Target Name="CreateOutputDirectories">

    <Task Name="MakeDir"
          Directories="$(OutputPath)"/>

  </Target>

  <Target Name="Build"
          DependsOnTargets="CreateOutputDirectories"
          Inputs="@(Compile)"
          Outputs="$(OutputAssembly)">

    <Task Name="Csc"
          Sources="@(Compile)"
          EmitDebugInformation="$(DebugSymbols)"
          Optimize="$(Optimize)"
          TargetType="$(OutputType)"
          OutputAssembly="$(OutputAssembly)"/>

  </Target>

</Project>
```

Notice that the `Build` target has two additional attributes: `Inputs` and `Outputs`. These attributes tell MSBuild what files this target expects as its inputs, and what files this target will generate as its output. At runtime, MSBuild examines the timestamps of the input and output files. If the input files are more recent than the output, this target will execute, otherwise this target will be skipped.[11]

Using file timestamps to determine whether a target runs is a reasonable solution. Tools like [n]make have long used this approach to determine whether a target needs recompilation, Tools like [N]Ant, however, leave it up to individual tasks to determine whether they need to

[10] Full builds are typically used in daily / integration builds.

[11] Since there can be multiple input and output files, if the oldest output is older than the newest input the target is rebuilt.

run. The MSBuild team chose to provide dependency analysis at the target level, while still leaving open the option for individual task authors to perform their own dependency analysis.[12]

Multiple Projects

Until now, we have only considered stand-alone projects; the only dependencies that our projects have had are on the assemblies that ship with the .NET Frameworks. In this example, we'll look at a project that has a dependency on another project.

Our new App project has a dependency on a Lib project. The Lib project creates a library assembly, lib.dll, which is referenced by the App project's assembly, app.exe. These two projects can be found in the App and Lib subdirectories. Here's the code in the lib.cs file:

```
public class Calc
{
  public static int Add(int x, int y)
  {
    return x + y;
  }
}
```

Here's the code in app.cs that uses the code in lib.cs:

```
using System;

public class App
{
  public static void Main()
  {
    int result = Calc.Add(3, 4);
    Console.WriteLine("The answer is: {0}", result);
  }
}
```

Since these projects are independent, they must be able to be built independently. Let's first look at lib.proj:

```
<Project DefaultTargets="Build">

  <PropertyGroup>
    <Property AssemblyName="Lib"/>
    <Property OutputType="library"/>
  </PropertyGroup>

  <ItemGroup>
    <Item Type="Compile" Include="Lib.cs"/>
  </ItemGroup>

  <Import Project="..\common.targets"/>
```

[12] There are cases where dependency analysis based on *file* timestamps is not adequate. For example, consider the case where you want to only upload files that have been recompiled to your FTP server. Since FTP servers aren't mapped to your local file system in Windows, file timestamps can't be used to perform the dependency analysis.

```
</Project>
```

As you would expect, this is a simple project file. The `Compile` Item references the `lib.cs` source file, and the `OutputType` is a Library assembly.

Things start to get interesting in the `App.proj` file. There are two new things that this project file must do. The first is the addition of a reference to the `lib.dll` assembly that is created by the `Lib` project. The second is a new target, `CopyProjectReferences`, which copies the `lib.dll` assembly to the `App` project's output directory. Since `lib.dll` is a private assembly, it must (by default) be in the same directory as `app.exe`.

The first thing to look at is the reference to `lib.dll`. In our project file, references to external assemblies are stored in the `ReferencePath` Item:

```
  <ItemGroup Condition=" '$(Configuration)' == 'Debug'
">
    <Item Type="ReferencePath"
          Include="..\lib\bin\Debug\Lib.dll"/>
    <Item Type="SourceFiles"
          Include="..\lib\bin\Debug\Lib.*"/>
  </ItemGroup>

  <ItemGroup Condition=" '$(Configuration)' == 'Release'
">
    <Item Type="ReferencePath"
          Include="..\lib\bin\Release\Lib.dll"/>
    <Item Type="SourceFiles"
          Include="..\lib\bin\Release\Lib.*"/>
  </ItemGroup>
```

Notice that the `ReferencePath` Item must be conditionally defined by an `ItemGroup` element since the location of `lib.dll` changes based on the build configuration. The `ReferencePath` Item is passed to the `References` attribute of the updated `csc` task in the `common.targets` file:

```
    <Task Name="Csc"
          Sources="@(Compile)"
          EmitDebugInformation="$(DebugSymbols)"
          Optimize="$(Optimize)"
          TargetType="$(OutputType)"
          References="@(ReferencePath)"
          OutputAssembly="$(OutputAssembly)"/>
```

The second task new task that the project file does is copy the referenced assemblies to the same directory as `app.exe`. This is done by the `CopyProjectReferences` target:

```
  <Target Name="CopyProjectReferences">
    <Task Name="Copy"
          SourceFiles="@(SourceFiles)"
          DestinationFiles="@(SourceFiles-
>'$(OutputPath)%n%x')"/>
  </Target>
```

This target uses the `Copy` task to copy the files referenced by the `SourceFiles` Item to the `App` project's output

directory.[13] The `Copy` task takes two parameters: a list of source files to copy in the `SourceFiles` attribute, and a list of the destination filenames in the `DestinationFiles` attribute. The `SourceFiles` Item is conditionally defined above, since the location of the outputs of the `Lib` project vary based on the project's configuration.

Notice the unusual syntax used to define the `DestinationFiles` attribute. This is an example of an *item transformation*, which tells MSBuild how to transform a filename. This transformation extracts the filename (%n)[14] and extension (%x) from the source filename, and appends them to the end of the `$(OutputPath)` property. This tells MSBuild where to put the files that are copied. This is a very powerful mechanism that lets you modify the destination filename and path based on property values.[15]

While this project file gets the job done, you can easily see a number of problems with it. First, `lib.dll` must already exist. Ideally, we would like to be able to build the `Lib` project from within the `App` project if it hasn't been built already. Second, there is a dependency on the name of the assembly generated by the `Lib` project. If you

[13] You may be wondering why I didn't choose to use the information contained already in the `ReferencePath` Item. This is due to the fact that debug builds generally create PDB files which also must be copied to the correct place in order for debugging to work correctly.

[14] This syntax will change by Beta 1. The replacement names at the time of this writing are %(FileName) and %(Extension).

[15] You could imagine appending the version number of an assembly to its filename after copying it, which is actually a good thing to do since it provides additional documentation to systems administrators.

change the assembly name in `lib.proj`, you will break the App project. In the next step, we will see some of the powerful features available in MSBuild that eliminates these problems.

Project Composition

In this step, I will greatly simplify the `app.proj` file by using the powerful project composition features of MSBuild. One of the key features of MSBuild is its ability to build another project from within an existing project. You can use the `MSBuild` task to invoke MSBuild on another project file. Furthermore, MSBuild makes it possible for a sub-project to send information back to the calling project. This makes it possible to *compose* a build by using project files from several projects.

In the previous step, we did two things: added a reference to `lib.dll`, and copied it to the directory that we are compiling `app.exe` into. To keep things simple, in this step I'll only implement a target called `BuildProjectReferences`. I'll defer implementing the `CopyProjectReferences` target until the next step.

In this step, the App project will call the `Lib` sub-project. A new Item, called `ProjectReferences`, is used to specify the location of the sub-projects that must be built prior to this project:

```
<ItemGroup>
  <Item Type="Compile"
        Include="App.cs"/>
  <Item Type="ProjectReference"
        Include="..\Lib\Lib.proj"/>
</ItemGroup>
```

The `common.targets` file has been updated to include a new Target called `BuildProjectReferences`:

```
<Target Name="BuildProjectReferences">

  <Task Name="MSBuild"
        Projects="@(ProjectReference)">

    <OutputItem TaskParameter="TargetOutputs"
                Type="ReferencePath"/>

  </Task>

</Target>
<!-- #region BuildProjectReferences -->

<!-- #region BuildTarget -->
<Target Name="Build"
        Inputs="@(Compile)"

Outputs="$(MSBuildProjectDirectory)$(OutputAssembly)"
        DependsOnTargets="BuildProjectReferences;
                          CreateOutputDirectories">
<!-- #endregion BuildTarget -->

  <Task Name="Csc"
        Sources="@(Compile)"
        EmitDebugInformation="$(DebugSymbols)"
        Optimize="$(Optimize)"
```

```
            TargetType="$(OutputType)"
            References="@(ReferencePath)"
            OutputAssembly="$(OutputAssembly)"/>

    </Target>

</Project>
```

As you can see, this Target uses the `MSBuild` task to build all of the project files referenced by the `ProjectReferences` Item[16]. The MSBuild task runs once for each project file in the `ProjectReferences` Item.

One of the challenges from the previous step was identifying the assemblies that are created when we run the referenced project. Fortunately, MSBuild has a feature that makes it possible for a referenced project to return information to the caller. When the `MSBuild` task runs, it executes at least one target in the referenced project. If that target defines its `Outputs`, the Item that describes the target's `Outputs` is returned to the caller's `MSBuild` task. In the caller's build script, you can see the `Outputs` of the referenced project by inspecting the `MSBuild` task's `TargetOutputs` property.

In this step, I now define the `Outputs` of a project's default target. If you look at `common.targets`, you can see that the `Build` target defines its `Outputs` now:

```
  <Target Name="Build"
          Inputs="@(Compile)"
Outputs="$(MSBuildProjectDirectory)$(OutputAssembly)"
          DependsOnTargets="BuildProjectReferences;
                            CreateOutputDirectories">
```

The `Outputs` of the target contains the path to the assembly generated by the `csc` task. The path is obtained by combining the `OutputAssembly` and `MSBuildProjectDirectory` properties.[17]

When `app.proj` calls `lib.proj` using the MSBuild task, the path to `lib.dll` is returned via the `TargetOutputs` property. The path to `lib.dll` is

[16] The MSBuild task runs the MSBuild engine in-process against each of the project files listed in its `Projects` attribute.

[17] This results in an absolute path since the `MSBuildProject` property identifies the directory that contains the project file that is currently being run.

important since we're going to need to tell the `csc` task where to find `lib.dll` when it compiles `app.exe`.

In the previous step, I used an Item called `ReferencePath` to store the list of referenced assemblies. This item contained the hard-coded path to `lib.dll`. In this step, I avoid hard-coding the path to `lib.dll`. Instead, I take the path to `lib.dll` that is returned by `TargetOutputs` and assign it to the `ReferencePath` Item by way of the `OutputItem` element:

```
  <Target Name="BuildProjectReferences">

    <Task Name="MSBuild"
          Projects="@(ProjectReference)">

      <OutputItem TaskParameter="TargetOutputs"
                  Type="ReferencePath"/>

    </Task>

  </Target>
  <!-- #region BuildProjectReferences -->

  <!-- #region BuildTarget -->
  <Target Name="Build"
          Inputs="@(Compile)"

Outputs="$(MSBuildProjectDirectory)$(OutputAssembly)"
          DependsOnTargets="BuildProjectReferences;
                            CreateOutputDirectories">
  <!-- #endregion BuildTarget -->

    <Task Name="Csc"
          Sources="@(Compile)"
          EmitDebugInformation="$(DebugSymbols)"
          Optimize="$(Optimize)"
          TargetType="$(OutputType)"
          References="@(ReferencePath)"
          OutputAssembly="$(OutputAssembly)"/>

  </Target>

</Project>
```

In this step, I've shown you how you can call one project from another. The referenced project doesn't know anything about the caller, yet is able to tell the caller about what it built. In the next step, we'll finish the job by implementing the `CopyProjectReferences` target.

Copying Referenced Projects

The `CopyProjectReferences` target copies all the files that were created by a referenced project to the current project's output directory. It uses the information in the `ReferencePath` Item to infer the names of the files that should be copied:

```
  <Target Name="CopyProjectReferences"
          Inputs="@(ReferencePath);@(ReferencePath-
>'%d%p%n.pdb')"
          Outputs="@(ReferencePath-
>'$(OutputPath)%n%x');@(ReferencePath-
>'$(OutputPath)%n.pdb')">

    <Task Name="Copy"
          SourceFiles="@(ReferencePath)"
          DestinationFiles="@(ReferencePath-
>'$(OutputPath)%n%x')"/>

    <Task Name="Copy"
          Condition="Exists('@(ReferencePath-
>'%d%p%n.pdb')')"
          SourceFiles="@(ReferencePath->'%d%p%n.pdb')"
```

```
        DestinationFiles="@(ReferencePath-
>'$(OutputPath)%n.pdb')"/>

  </Target>
```

For each referenced project, there are potentially two files that need to be copied: the generated assembly and its associated `.pdb` file.[18] However, since the `.pdb` file may or may not be present (it is not in Release configurations) the Copy task might fail if it can't find the `.pdb` file. This is why the second Copy task sets the `ContinueOnError` attribute to true so that failure to find the `.pdb` file does not abort the build.[19]

A VB.NET Project

So far, we've only considered projects that are written using C#. What happens when we want to add a VB.NET project? In this step, I'll show you how you can modify our `.targets` files so that we can effect this change.

The first step is to look at the existing common.targets file to see what code is specific to the C# compiler. It doesn't take long to see that the only piece of code is the `csc` Task:

```
<Task Name="Csc"
      Sources="@(Compile)"
      EmitDebugInformation="$(DebugSymbols)"
      Optimize="$(Optimize)"
      TargetType="$(OutputType)"
      References="@(ReferencePath)"
      OutputAssembly="$(OutputAssembly)"/>
```

To solve this problem, I'm going to introduce a new target called `Compile`. It contains the language-specific call to the appropriate task. There will be two new `.targets` files that I've created for this step. The first is `csharp.targets`:

[18] This is a simplified view of the real problem. The assemblies to be copied could refer to other private assemblies that need to be copied to the target directory.So the real answer is that a recursive copy is required.

[19] I'm not saying that `ContinueOnError` is necessarily a good thing, but it's a reasonable solution if an error is not important to the overall success/failure of the build. If the error is important to the overall success/failure of a build then some kind of exception handling model is called for, The Ant community has created a `try` task to handle these types of scenarios.

```
<Project>

  <Target Name="Compile"
          Inputs="@(Compile)"
          Outputs="$(OutputAssembly)">

    <Task Name="Csc"
          Sources="@(Compile)"
          EmitDebugInformation="$(DebugSymbols)"
          Optimize="$(Optimize)"
          TargetType="$(OutputType)"
          References="@(ReferencePath)"
          OutputAssembly="$(OutputAssembly)"/>

  </Target>

  <Import Project="common.targets"/>

</Project>
```

The second is `vb.targets`:

```
<Project>

  <Target Name="Compile"
          Inputs="@(Compile)"
          Outputs="$(OutputAssembly)">

    <Task Name="Vbc"
          Sources="@(Compile)"
          EmitDebugInformation="$(DebugSymbols)"
          Optimize="$(Optimize)"
          TargetType="$(OutputType)"
          References="@(ReferencePath)"
          OutputAssembly="$(OutputAssembly)"/>

  </Target>

  <Import Project="common.targets"/>

</Project>
```

Our individual application project files will import either `csharp.targets` or `vb.targets` as appropriate:

```
<Project DefaultTargets="Build">

  <PropertyGroup>
    <Property AssemblyName="Lib"/>
    <Property OutputType="library"/>
  </PropertyGroup>

  <ItemGroup>
    <Item Type="Compile" Include="Lib.vb"/>
  </ItemGroup>

  <Import Project="..\vb.targets"/>

</Project>
```

Notice that each of these targets files imports the `common.targets` file. Notice that I've also modified the `Build` target to introduce a dependency to the new `Compile` target:

```
  <Target Name="Build"

Outputs="$(MSBuildProjectDirectory)$(OutputAssembly)"
          DependsOnTargets="BuildProjectReferences;
                            CreateOutputDirectories;
                            CopyProjectReferences;
                            Compile"/>
```

By organizing our code this way, we have localized all language-specific tasks to a single file. If we want to support other languages like C++ down the road, all we need to do is create a new `.targets` file for that language; we don't need to change any of our existing `.targets` files.

VS.NET .csproj Walk Through

In the first ten steps of this guide, I walked you through an implementation of the core elements of a build system. Along the way, I've shown you many of the features of

MSBuild that make it easier to create complex build scripts. However, most developers are likely to just use the core build scripts that ship with MSBuild. In this final step of the guide, I'll walk you through a sample project file that uses the MSBuild core build scripts:

```
<Project DefaultTargets="Build">

  <PropertyGroup>
    <Property AssemblyName="App"/>
    <Property OutputType="exe"/>
  </PropertyGroup>

  <ItemGroup>
    <Item Type="Compile" Include="App.cs"/>
    <Item Type="ProjectReference"
Include="..\Lib\Lib.proj"/>
  </ItemGroup>

  <Import Project="..\common.properties"/>
  <Import Project="$(MSBuildBinPath)\csharp.targets"/>

</Project>
```

Compare this file with `app.proj` from the previous step:

```
<Project DefaultTargets="Build">

  <PropertyGroup>
    <Property AssemblyName="App"/>
    <Property OutputType="exe"/>
  </PropertyGroup>

  <ItemGroup>
    <Item Type="Compile" Include="App.cs"/>
    <Item Type="ProjectReference"
Include="..\Lib\Lib.proj"/>
  </ItemGroup>

  <Import Project="..\csharp.targets"/>

</Project>
```

As you can see, the central elements of these two files are identical. The differences lie in the project files that are imported. I intentionally designed the walk-through so that it would duplicate the names of all properties and items that are used by the MSBuild core scripts.

The key file is the `csharp.targets` file.[20] This file is found in the `MSBuildBinPath` directory, which on my computer is mapped to `\Windows\Microsoft.Net\Framework\V1.2.30703\`. When you examine this file, you'll see that it imports another file called `framework.targets`, which is analogous to the `common.targets` file that we created earlier.

Inside this file, you'll find a `Compile` target, which invokes the `csc` task, as you would expect. You'll immediately notice, however, that far more attributes of

[20] VB.NET projects will use another file called `visualbasic.targets`.

the `csc` task are used in this target. This reflects the fact that there are many command-line switches for the C# compiler.

Next, you should open the `framework.targets` file. The first thing that you'll notice is that there are a large number of pre-defined properties at the top of the file.[21] You'll also find a number of targets that should be familiar to you, such as `CopyProjectReferences`[22] and `Build`. There are a number of other targets that handle features that are exposed by the VS.NET IDE, such as `PreBuildEvent` and `PostBuildEvent`.[23]

The important thing to remember while exploring the `.targets` files that ship with MSBuild is how most of the complexity of the build system is hidden from you. The key to this extensibility is the ability to compose a new project from several existing project files. You could imagine that third parties could provide additional `.targets` files that provide additional features. For example, you could imagine a `.targets` file that implements a target called `CheckOut` which goes out and retrieves the latest set of source code files from your version control system.

---

[21] In the walk-through, I deliberately partitioned the default properties into a separate file: `common.properties`. This was done to avoiding mixing code and data inside the `common.targets` file.

[22] You'll see from the comments in the file that one of the challenges of `CopyProjectReferences` is to *recursively* copy all required assemblies from any referenced projects in the referenced project.

[23] One of the major challenges of the MSBuild team is to ensure that MSBuild projects can duplicate the semantics of the existing VS.NET build system. Over the upcoming weeks and months you'll find additional coverage on my weblog (http://www.iunknown.com) about specific challenges.

Another important thing to remember is the fact that MSBuild is currently in a state of flux. The file format will most definitely change between the PDC version and Beta 1. These changes will be breaking changes, and you will need to make extensive changes to your build scripts when Beta 1 ships if you choose to start writing build scripts today. That said, however, I would spend time working through the samples in this guide. Think about the project composition features – they're more powerful than they appear at first glance. Think about how they can be used to modularize your existing build system. Finally, make sure that you send comments and feedback to me: jlam@iunknown.com. I'll make sure that your comments make it through to the right folks.

Final Frame



It was our one-year wedding anniversary this past August, so we rented a cottage in Muskoka to spend a week away from it all. One evening, a spectacular purplish sunset presented appeared ...