2

Building a Longhorn Application

To build a Longhorn application, you need the Longhorn Software Development Kit (SDK) installed. Alternatively, you can install a Microsoft Visual Studio release that supports Longhorn. In this book, I don't discuss using Visual Studio because its wizards, fancy code generation features, and project build capabilities obscure what actually happens under the covers. I believe that you should understand what a tool does for you before you rely on the tool.

When you install the Longhorn SDK, it creates a set of Start menu items that you can use to create a command prompt session in which you can build Longhorn applications. To build Debug versions of your application on a Microsoft Windows XP 32-bit system, navigate through the following menu items to create the appropriate command prompt session:

- Start
- Programs
- Microsoft Longhorn SDK
- Open Build Environment Window
- Windows XP 32-bit Build Environment
- Set Windows XP 32-bit Build Environment (Debug).

The Microsoft .NET Build Engine—MSBuild.exe

MSBuild is the primary tool you use to build a Longhorn application. You can run MSBuild with the help command-line option to get detailed information on its usage:

```
MSBuild /?
```

When you execute MSBuild without any command-line arguments, as shown here, it searches in the current working directory for a file name that ends with "proj", for example, .proj, .csproj, etc. When it finds one, it builds the project according to the directives in that file.

```
MSBuild
```

When you have more than one project file in the directory, you can specify the appropriate project file on the command line:

```
MSBuild <ProjectName>.proj
```

Normally, MSBuild builds the default target in the project file. You can override this and specify the target you want build. For example, to build the target named *CleanBuild*, you invoke MSBuild as follows:

```
MSBuild /t:Cleanbuild
```

Building Hello World Using MSBuild

Let's look at the files necessary to create a simple navigation-based Hello World application. Later I'll describe the purpose and use of each file in detail.

First, you need to define the *Application* object. You do this in a file typically called the *application definition file*. This HelloWorldApplication.xaml file defines my *Application* object.

HelloWorldApplication.xaml

```
<NavigationApplication xmlns="http://schemas.microsoft.com/2003/xaml"
                       StartupUri="HelloWorld.xaml" />
```

This definition says, "For my *Application* object, I want to use an instance of the *MSAvalon.Windows.Navigation.NavigationApplication* class. On startup, the application should navigate to and display the user interface (UI) defined in the HelloWorld.xaml file."

Here are the contents of the HelloWorld.xaml file. It's a slightly more interesting version of the previous Hello World example in Chapter 1.

HelloWorld.xaml

```
<Border xmlns="http://schemas.microsoft.com/2003/xaml">
  <FlowPanel>
    <SimpleText Foreground="DarkRed" FontSize="14">Hello World!</SimpleText>
  </FlowPanel>
</Border>
```

Now that I have all the "code" for my simple Hello World application, I need a project file that defines how to build my application. Here's my HelloWorld.proj file.

HelloWorld.proj

```
<Project DefaultTargets="Build">
  <PropertyGroup>
    <Property Language="C#" />
    <Property DefaultClrNameSpace="IntroLonghorn" />
    <Property TargetName="HelloWorld" />
  </PropertyGroup>

  <!--Imports the target which contains all the common targets-->
  <Import Project="$(LAPI)\WindowsApplication.target" />

  <ItemGroup>
    <!-- Application markup -->
    <Item Type="ApplicationDefinition" Include="HelloWorldApplication.xaml" />

    <!-- Compiled Xaml Files list -->
    <Item Type="Pages" Include="HelloWorld.xaml"/>
  </ItemGroup>
</Project>
```

Put these three files in a directory. Open a Longhorn SDK command prompt, navigate to the directory containing your files, and run MSBuild. It will compile your program into an executable.

We'll examine the contents of the application definition file later in this chapter. In Chapter 3, I describe in detail many of the Extensible Application Markup Language (XAML) elements you can use to define a UI. Before we look at the project file in more depth, let's review some MSBuild terminology.

MSBuild Terminology

Let's establish definitions for some MSBuild elements. A *Property* is a key-value pair. A *Property*'s value can originate from an environment variable,

from a command-line switch, or from a *Property* definition in a project file, as shown here:

```
<Property OutputDir="bin\" />
```

You can think of an *Item* as an array of files. An *Item* can contain wildcards and can exclude specific files. MSBuild uses the *Type* attribute of an *Item* to categorize items, as shown here:

```
<Item Type="Compile" Include="*.cs" Exclude="DebugStuff.cs" />
```

A *Task* is an atomic unit in the build process. A *Task* can accept input parameters from *Property* elements, *Item* elements, or plain strings. The *Name* of a *Task* identifies the build .NET data type required to perform the *Task*. A *Task* can emit *Item*s that other *Task*s consume. MSBuild includes many tasks, all of which can be broadly categorized as

- .NET tool tasks
- Deployment tasks
- Shell tasks

For example, the *Task* with a *Name* of *Csc* invokes the C# compiler as the build tool, which compiles all *Item* elements specified in the *Sources* attribute (which specifies *Item* elements with a *Type* of *Compile*) into an assembly, and produces the assembly as an output *Item*.

```
<Task Name="Csc" AssemblyName="$(OutputDir)\HelloWorld.exe"
                 Sources="@(Compile)" />
```

A *Target* is a single logical step in the build process. A *Target* can perform timestamp analysis. This means that a *Target* will not run if it's not required. A *Target* executes one or more *Task*s to perform the desired operations, as shown here:

```
<Target Name="CopyToServer"
        Inputs="$(OutputDir)\HelloWorld.exe"
        Outputs="\\DeployServer\$(BuildNum)\HelloWorld.exe"
        DependsOnTargets="Compile">

  <Task Name="Copy" ... />
</Target>
```

A *Condition* attribute is roughly equivalent to a simple *if* statement. A *Condition* can compare two strings or check for the existence of a file or directory. You can apply a *Condition* to any element in a project file. For example, here's a group of properties that are defined only when the *Configuration* property has the value *Debug*:

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

An *Import* is roughly equivalent to a C/C++ *#include* statement, as shown in the following example. When you import a project, the contents of the imported project logically become a part of the importing project.

```
<Import Project="$(LAPI)\WindowsApplication.target" />
```

Now that the terminology is out of the way, let's examine a typical project file.

Building a Longhorn Executable Application

Here's a simple, but relatively comprehensive, project file that builds an executable Longhorn application:

```
<Project DefaultTargets="Build">
  <PropertyGroup>
    <Property Language="C#" />
    <Property DefaultClrNameSpace="IntroLonghorn" />
    <Property TargetName="MyApp" />
  </PropertyGroup>

  <Import Project="$(LAPI)\WindowsApplication.target" />

  <ItemGroup>
    <Item Type="ApplicationDefinition" Include="MyApp.xaml" />

    <Item Type="Pages" Include="HomePage.xaml" />
    <Item Type="Pages" Include="DetailPage.xaml" />
    <Item Type="Code" Include="DetailPage.xaml.cs"/>

    <Item Type="DependentProjects" Include="MyDependentAssembly.proj" />

    <Item Type="Components" Include="SomeThirdParty.dll" />

    <Item Type="Resources" Include="Picture1.jpg"
          FileStorage="embedded" Localizable="False"/>
```

```
    <Item Type="Resources" Include="Picture2.jpg"
          FileStorage="embedded" Localizable="True"/>
  </ItemGroup>
</Project>
```

The *Project* Element

All project files begin with a root element definition named *Project*. Its *DefaultTargets* attribute specifies the names of the targets that the system should build when you don't otherwise specify a target. In this example, I specify that, by default, the system should build the target named *Build*.

The *PropertyGroup* and *Property* Elements

Build rules can conditionally execute based on property values. As mentioned, a property's value can originate from an environment variable, from an MSBuild command-line switch, or from a property definition in a project file.

A project for an application must specify, at a minimum, a value for the *Language* and *TargetName* properties. In this example, I specify that the language is C# and that the name of the resulting application should be *MyApp*. I've also assigned a value to the property named *DefaultClrNameSpace*.

The build system compiles each XAML file into a managed class definition. By default, the managed class will have the same name as the base file name of the XAML source file. For example, the file Markup.xaml compiles into a definition of a class named *Markup*. By setting the *DefaultClrNameSpace* property to *IntroLonghorn*, I'm asking the build system to prefix generated class names with the *IntroLonghorn* namespace. Because of this, the build system produces a class named *IntroLonghorn.Markup* for the Markup.xaml definition.

I defined my properties prior to importing other projects, so the rules in the imported projects will use my specified property values—for example, I'll get the proper build rules for C# applications because I define the *Language* property as *C#*.

The *Import* Element

The rules in the *Build* target produce my Longhorn application's executable file. Specifying those build rules in every project file would be tedious and repetitive. So a little later in the project file, I use the following definition to import a predefined project file named *WindowsApplication.target*:

```
<Import Project="$(LAPI)\WindowsApplication.target" />
```

This imported file contains the standard build rules for building a Windows application, and it (indirectly) defines the target named *Build*.

The *ItemGroup* and *Item* Elements

The *ItemGroup* element and its child *Item* elements define all the parts required to build the application.

You must have one *Item* with a *Type* of *ApplicationDefinition*, as shown here. This *Item* specifies the file that describes the *Application* object to use for your application. The *Application* object is typically an instance of either the *MSAvalon.Windows.Application* class or the *MSAvalon.Windows.Navigation.NavigationApplication* class, both of which I describe later in this chapter.

```
<Item Type="ApplicationDefinition" Include="MyApp.xaml" />
```

Each *Item* with a *Type* of *Pages* defines a set of XAML files, as shown here. The build system compiles these XAML definitions into classes that it includes in the resulting assembly.

```
<Item Type="Pages" Include="HomePage.xaml" />
<Item Type="Pages" Include="DetailPage.xaml" />
```

Each *Item* with a *Type* of *Code* represents a source file, as shown here. The build system compiles these source files using the appropriate compiler selected by your project's *Language* property.

```
<Item Type="Code" Include="DetailPage.xaml.cs"/>
```

This project might depend on other projects. The build system must compile these dependent projects before it can build this project. You list each such dependent project using an *Item* with *Type* of *DependentProjects*:

```
<Item Type="DependentProjects" Include="MyDependentAssembly.proj" />
```

Code in this project might use types in a prebuilt assembly, also known as a *component assembly*. To compile code using such component assemblies, the compiler needs a reference to each assembly. In addition, when you deploy your application, you will need to deploy these component assemblies as well. You list each component assembly using an *Item* with *Type* of *Components*:

```
<Item Type="Components" Include="SomeThirdParty.dll" />
```

A referenced assembly is somewhat different from a component assembly. In both cases, your code uses types in a prebuilt assembly. However, you don't ship a referenced assembly as part of your application, whereas you do ship a component assembly as part of your application. The build system needs to know this distinction.

You specify an *Item* with a *Type* of *References* to indicate that the compiler must reference the specified assembly at build time, as shown here, but the assembly will not be part of the application deployment. The build system automatically includes references to standard system assemblies—for example, mscorlib.dll, System.dll, PresentationFramework.dll. and more—but you'll have to add any nonstandard assembly your application must reference.

```
<Item Type="References" Include="SharedThirdParty.dll" />
```

Your application might also use resources. An *Item* with a *Type* of *Resources* describes a resource used by the application, as shown here. The build system can embed the resource into the resulting assembly or include it as a stand-alone file. The build system can also place localizable resources into satellite assemblies.

```
<Item Type="Resources" Include="Picture1.jpg"
      FileStorage="embedded" Localizable="False"/>
<Item Type="Resources" Include="Picture2.jpg"
      FileStorage="embedded" Localizable="True"/>
```

Building a Longhorn Library Assembly

You'll also want to build libraries in addition to executable applications. The primary differences between an application project and a library project are these:

- A library project sets the value of the *TargetType* property to *Library*.
- A library project typically does not include an application definition item.

Here's an example of a project file that creates a library:

```
<Project DefaultTargets="Build">
  <PropertyGroup>
    <Property Language="C#" />
    <Property DefaultClrNameSpace="IntroLonghorn" />
    <Property TargetName="MyLibrary" />
    <Property TargetType="Library" />
  </PropertyGroup>

  <Import Project="$(LAPI)\WindowsApplication.target" />

  <ItemGroup>
    <Item Type="Pages" Include="ErrorPage.xaml" />
    <Item Type="Code" Include="ErrorPage.xaml.cs"/>
    <Item Type="Code" Include="Utilities.cs"/>
```

```
    <Item Type="DependentProjects" Include="MyDependentAssembly.proj" />

    <Item Type="Components" Include="SomeThirdParty.dll" />

    <Item Type="Resources" Include="Picture1.jpg"
          FileStorage="embedded" Localizable="False"/>
    <Item Type="Resources" Include="Picture2.jpg"
          FileStorage="embedded" Localizable="True"/>
  </ItemGroup>
</Project>
```

Building a Longhorn Document

You aren't restricted to building applications with XAML. You can also use XAML files to create a highly interactive, intelligently rendered, adaptive document for a user to read. In this case, your XAML files collectively represent pages of a document. You can use the MSBuild engine to build such documents.

The changes to the project file to build a document instead of an application are minor:

- Set the value of the *TargetType* property to *Document*.
- Import the WindowsDocument.target project for the appropriate build rules.
- Include all other project files as usual.

It's important to understand what a *TargetType* of *Document* really produces. When you build a *Document*, the build output is a .container file, and the build system optimizes the contents of the container for download rather than speed. A container file is an extension of the Windows Structured Storage, also known as DocFile, format. Longhorn container handling provides features that allow rendering of partially downloaded files. Therefore, you don't need the entire container downloaded before the application starts running.

In addition, when you ask MSBuild to create a container file, it compiles each XAML file into a binary representation of the XML, called binary XAML (BAML). BAML is far more compact than either the original text file or a compiled-to-IL assembly. BAML files download more quickly—are optimized for download—but an interpreter must parse them at run time to create instances of the classes described in the file. Therefore, such files are not optimized for speed. Up to now, I've been generating compiled-to-IL files (also known as CAML files, short for compiled XAML).

Here's an example of a project file that creates an electronic document:

```
<Project DefaultTargets="Build">
  <PropertyGroup>
    <Property TargetType="Document" />
      <Property Language="C#" />
      <Property DefaultClrNameSpace="IntroLonghorn" />
      <Property TargetName="MyDocument" />
  </PropertyGroup>

  <Import Project="$(LAPI)\WindowsDocument.target" />

  <ItemGroup>
    <Item Type="ApplicationDefinition" Include="MyApp.xaml" />

    <Item Type="Pages" Include="Markup.xaml" />
    <Item Type="Pages" Include="Navigate.xaml" />
    <Item Type="Code" Include="Navigate.xaml.cs"/>

    <Item Type="Resources" Include="Picture1.jpg"
          FileStorage="embedded" Localizable="False"/>
    <Item Type="Resources" Include="Picture2.jpg"
          FileStorage="embedded" Localizable="True"/>
  </ItemGroup>
</Project>
```

Now that you've learned how to build the various types of Longhorn applications and components, let's look at XAML files in more detail. Specifically, let's look at what the build system does when it turns a XAML file into a .NET class.

A XAML File as a Class Declaration

The application definition file is the XAML file that defines the class of your application's *Application* object. However, in general, a XAML document is simply a file that defines a class. The class produced by the XAML definition derives from the class associated with the XML document's root element name. By default, the build system uses the XAML base file name as the generated class name.

Creating an Application Definition File for a Navigation Application

Recall that the *Item* element with *Type* of *ApplicationDefinition* specifies the name of the XAML file that defines the *Application* object. In other words, this element specifies the XAML file that contains the entry point for your applica-

tion. The Longhorn platform will create an instance of the *MSAvalon.Windows.Application*-derived type that you define in this file and let it manage the startup, shutdown, and navigation of your application.

In Chapter 1, you saw how to create and use an application instance programmatically. The following XAML file uses markup to define the *Application* object for a project:

```
<NavigationApplication xmlns="http://schemas.microsoft.com/2003/xaml"
                       StartupUri="HelloWorld.xaml" />
```

I expect that most Longhorn applications will be navigation-based applications and, therefore, will often just reuse the standard *NavigationApplication* object. You can reuse this application definition file for most of your navigation-based applications by changing only the value of the *StartupUri* attribute.

For example, if the previous application definition resides in the HelloWorldApplication.xaml file and I use the HelloWorld.proj project file listed previously, the build system produces the following class declaration:

```
namespace IntroLonghorn {
  class HelloWorldApplication :
           MSAvalon.Windows.Navigation.NavigationApplication {
⋮
   }
 }
```

The namespace results from the *DefaultClrNameSpace* declaration in the project file, the declared class name is the same as the base file name, and the declared class extends the class represented by the root element in the XAML file.

Customizing the Generated Code Using Attributes

When you declare a root element in a XAML file, you can use attributes on the root element to control the name of the generated class declaration. You can use any of the following optional attributes:

- A namespace prefix definition that associates a prefix with a namespace named *Definition*. You must define a prefix for this namespace to use the *Language* and *Class* attributes. Traditionally, the *def* prefix is used.
- A *Language* attribute (defined in the *Definition* namespace) that specifies the programming language used by any inline code in the XAML file.

- A *Class* attribute (defined in the *Definition* namespace) that specifies the name of the generated class. When you specify a name containing one or more periods, the build system does not prefix the name with the *DefaultClrNameSpace* value.

As an example, let's change the contents of the HelloWorldApplication.xaml file to the following:

```
<NavigationApplication xmlns="http://schemas.microsoft.com/2003/xaml"
                       xmlns:def="Definition"
                       def:Class="Special.MyApp"
                       def:CodeBehind="HelloWorldApplication.xaml.cs"
                       StartupUri="HelloWorld.xaml" />
```

The generated class would then be this:

```
namespace Special {
  class MyApp :
           MSAvalon.Windows.Navigation.NavigationApplication {
⋮
  }
}
```

Using Code and Markup in the Same Class

Nearly all applications will require you to write some code—for example, a click event handler for a button or a virtual method override—in addition to the markup that specifies the UI. Recall from Chapter 1 that my non-navigation-based application overrode the *OnStartingUp* method to create its window and controls. I'll rewrite that example to illustrate how you would combine application code and markup.

While this forthcoming example creates a non-navigation application, I want to emphasize that there is really no compelling reason to create such an application. You can always create a navigation-based application that never actually navigates to a different page. However, writing such an application requires me to mix code and markup in the same class therefore provides a good example.

Recall that creating a non-navigation application requires you to define a custom class that inherits from *MSAvalon.Windows.Application* and that overrides the *OnStartingUp* method. The application definition file declares the application object class that your program uses. Therefore, a non-navigation application must define its overriding *OnStartingUp* method in the same class.

Except for the following changes, an application configuration file for a non-navigation application contains the same items as a definition file for a navigation application:

- The root element is *Application* instead of *NavigationApplication*.
- The file must contain or reference the implementation of the *OnStartingUp* method for your application class.

Because I need to use markup and code to implement a single class, I need to show you a technique to associate a source code file with a XAML file.

Associating a Source-Behind File with a XAML File

You will frequently want to develop portions of your application by using markup and to develop other parts by using a more traditional programming language. I strongly recommend separating the UI and the logic into individual source files by using the following technique.

You can add a XAML *CodeBehind* element (defined in the *Definition* namespace) to the root element of any XAML file and specify the name of a source code file (also known as the *code-behind file*). The build engine will compile the XAML declarations into a managed class. The build system also compiles the code-behind file into a managed class declaration. The tricky aspect is that both of these class declarations represent partial declarations of a single class.

Here's a XAML definition that produces a non-navigation application class equivalent to the first example in Chapter 1:

```
<Application xmlns="http://schemas.microsoft.com/2003/xaml"
             xmlns:def="Definition"
             def:Language="C#"
             def:Class="IntroLonghorn.CodeBehindSample"
             def:CodeBehind="CodeBehind.xaml.cs" />
```

There are two noteworthy aspects to this application definition file:

- The *Language* attribute specifies that the code-behind file contains C# source code.
- The *CodeBehind* attribute specifies that the source file name is CodeBehindMySample2.xaml.cs.

Here's the matching source-behind file:

```
namespace IntroLonghorn {
  using System;
  using MSAvalon.Windows;
```

```
using MSAvalon.Windows.Controls;
using MSAvalon.Windows.Media;

public partial class CodeBehindSample {
  MSAvalon.Windows.Controls.SimpleText txtElement;
  MSAvalon.Windows.Window              mainWindow;

  protected override
  void OnStartingUp (StartingUpCancelEventArgs e) {
    base.OnStartingUp (e);
    CreateAndShowMainWindow ();
  }

  private void CreateAndShowMainWindow () {
    // Create the application's main window
    mainWindow = new MSAvalon.Windows.Window ();

    // Add a dark red, 14 point, "Hello World!" text element
    txtElement = new MSAvalon.Windows.Controls.SimpleText ();
    txtElement.Text = "Hello World!";
    txtElement.Foreground = new
     MSAvalon.Windows.Media.SolidColorBrush (Colors.DarkRed);
    txtElement.FontSize = new FontSize (14,
                                        FontSizeType.Point);
    mainWindow.Children.Add (txtElement);
    mainWindow.Show ();
  }
 }
}
```

Notice the *partial* keyword in the class declaration in the code-behind file. This keyword states that the compiler should merge this class definition with other definitions of the same class. This allows you to provide multiple partial definitions of a class, each in a separate source file, that the compiler combines into a single class definition in the resulting assembly.

Mixing Source Code and Markup in a Single XAML File

I think it's just wrong to mix source code and markup in the same file. I even considered not showing you how to do it. However, some evildoer somewhere will write a sample program using this technique, so you might need to understand what he has done. Moreover, you can then use the code-behind file approach described previously to rid the world of a small amount of evil and separate the UI from the logic.

Here's an application definition file with the source code inserted directly inline with the markup:

```
<Application xmlns="http://schemas.microsoft.com/2003/xaml"
    xmlns:def="Definition"
    def:Language="C#"
    def:Class="IntroLonghorn.MySample2" >

  <def:Code>
  <![CDATA[
    protected override void OnStartingUp (StartingUpCancelEventArgs e) {
      base.OnStartingUp (e);
      CreateAndShowMainWindow ();
    }
    . . . Remaining methods elided for clarity
  ]]>
  </def:Code>
</Application>
```

In this example, the *Language* attribute specifies that the inline source code is C#. Notice that the *Code* element is a CDATA block containing the inline source code. It's sometimes technically necessary to enclose inline source code in an XML CDATA block to ensure that the document is well-formed. In fact, the XAML parser always requires you to enclose the inline source code in a CDATA block, even when omitting it produces a well-formed document.

I apologize once again for showing you such a travesty.

The Application Manifest

When you compile an application, MSBuild produces the .exe file plus two manifest files: the application manifest, *.manifest, and a deployment manifest, *.deploy. You use these manifests when deploying an application or document from a server. First, copy the application, all of its dependencies, and the two manifest files to the appropriate location on your server. Second, edit the deployment manifest so that it points to the location of the application manifest.

For completeness, let's look at examples of the application and deployment manifests. The application manifest, shown in the following example, is actually not as interesting as the deployment manifest. The application manifest simply defines all the parts that make up an application. MSBuild produces the application manifest when it builds your application, and you typically modify little or nothing in it.

HelloWorld.manifest

```
<?xml version="1.0" encoding="utf-8"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0"
          xmlns:asmv2="urn:schemas-microsoft-com:asm.v2"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:schemas-microsoft-com:asm.v1 assembly.adaptive.xsd">
```

```
<assemblyIdentity name="HelloWorld" version="1.0.0.0"
                  processorArchitecture="x86" asmv2:culture="en-us"
                  publicKeyToken="0000000000000000" />

<entryPoint name="main" xmlns="urn:schemas-microsoft-com:asm.v2"
            dependencyName="HelloWorld">

  <commandLine file="HelloWorld.exe" parameters="" />
</entryPoint>

<TrustInfo xmlns="urn:schemas-microsoft-com:asm.v2" xmlns:temp="temporary">
  <Security>
    <ApplicationRequestMinimum>
      <PermissionSet class="System.Security.PermissionSet" version="1"
                     ID="SeeDefinition">
        <IPermission
          class="System.Security.Permissions.FileDialogPermission"
          version="1" Unrestricted="true" />
        <IPermission
          class="System.Security.Permissions.IsolatedStorageFilePermission"
          version="1" Allowed="DomainIsolationByUser" UserQuota="5242880" />
        <IPermission
          class="System.Security.Permissions.SecurityPermission"
          version="1" Flags="Execution" />
        <IPermission
          class="System.Security.Permissions.UIPermission" version="1"
          Window="SafeTopLevelWindows" Clipboard="OwnClipboard" />
        <IPermission
          class="System.Security.Permissions.PrintingPermission"
          version="1" Level="SafePrinting" />
        <IPermission
          class="MSAvalon.Windows.AVTempUIPermission, PresentationFramework,
                 Version=6.0.4030.0, Culture=neutral,
                 PublicKeyToken=a29c01bbd4e39ac5" version="1"
                 NewWindow="LaunchNewWindows" FullScreen="SafeFullScreen" />
      </PermissionSet>

      <AssemblyRequest name="HelloWorld"
                       PermissionSetReference="SeeDefinition" />
    </ApplicationRequestMinimum>
  </Security>
</TrustInfo>

<dependency asmv2:name="HelloWorld">
  <dependentAssembly>
    <assemblyIdentity name="HelloWorld" version="0.0.0.0"
                      processorArchitecture="x86" />
  </dependentAssembly>
```

```
    <asmv2:installFrom codebase="HelloWorld.exe"
                       hash="5c58153494c16296d9cab877136c3f106785bfab"
                       hashalg="SHA1" size="5632" />
  </dependency>
</assembly>
```

Most of the contents of the application manifest should be relatively obvious. The *entryPoint* element specifies the name of the entry point method, *main*, and references the *dependency*, named *HelloWorld*, that contains the entry point. The *entryPoint* element also contains the program name and command-line argument that the shell will need to run the application.

The *HelloWorld dependency* element contains the information (the *dependentAssembly* element) that specifies the dependent assembly and an *installFrom* element that tells the loader where to find the assembly's file and the file's original hash. The loader can use the hash to detect changes made to the assembly subsequent to compilation.

The Longhorn Trust Manager uses the *TrustInfo* element to determine the security permissions that the application requires. In the previous example, my HelloWorld application defines a set of permissions it names the *SeeDefinition* permission set. Immediately after I define the set of permissions, the *AssemblyRequest* element requests that the assembly named *HelloWorld* receives at least the set of permissions in the set named *SeeDefinition*. The permissions in this example are the permissions normally granted to applications running in the SEE, so the Hello World application runs without displaying to the user any Trust Manager security warnings.

The Deployment Manifest

As mentioned, the deployment manifest is more interesting. The deployment manifest contains, obviously enough, settings that control the deployment of the application.

HelloWorld.deploy

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0"
          xmlns:asmv2="urn:schemas-microsoft-com:asm.v2"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="urn:schemas-microsoft-com:asm.v1 assembly.adaptive.xsd">

  <assemblyIdentity name="HelloWorld.deploy" version="1.0.0.0"
                    processorArchitecture="x86" asmv2:culture="en-us"
                    publicKeyToken="0000000000000000" />
```

```
  <description asmv2:publisher="Wise Owl, Inc."
               asmv2:product="Brent's HelloWorld Application"
    asmv2:supportUrl="http://www.wiseowl.com/AppServer/HelloWorld/support.asp"
  />

  <deployment xmlns="urn:schemas-microsoft-com:asm.v2"
              isRequiredUpdate="false">
    <install shellVisible="true" />
    <subscription>
      <update>
        <beforeApplicationStartup />
        <periodic>
          <minElapsedTimeAllowed time="6" unit="hours" />
          <maxElapsedTimeAllowed time="1" unit="weeks" />
        </periodic>
      </update>
    </subscription>
  </deployment>
  <dependency>
    <dependentAssembly>
      <assemblyIdentity name="HelloWorld" version="1.0.0.0"
                        processorArchitecture="x86" asmv2:culture="en-us"
                        publicKeyToken="0000000000000000" />
    </dependentAssembly>
    <asmv2:installFrom codebase="HelloWorld.manifest" />
  </dependency>
</assembly>
```

The deployment manifest contains information that Longhorn requires to install and update an application. Notice that the *assemblyIdentity* element of the deployment manifest references the application manifest. After all, the application manifest already describes all the components of an application. To install an application, the deployment manifest says, in effect, "Here's the description of the files you need to install this application."

Of course, when you install an application, you also need more information than just the files to copy onto a system. The *description* element lists the *publisher*, *product*, and *supportUrl* attributes; the system displays their contents in the Add/Remove Programs dialog box.

The *deployment* element specifies how to deploy and update the application after deployment. In this example, the application is visible to the shell, and the client's system will check for and, if necessary, download a new version of the application each time the user starts the application. In addition, the system periodically—no more than every six hours and no less than once a week—checks for a new version. When the periodic check locates a new version, Longhorn will download the new version in the background and install it; it will then be ready to run the next time the user executes the application.

Running the Application

Normally, a user will "run" the application manifest to execute the application from the server directly without installing the application on the local computer. Longhorn downloads the components of the application as needed. In this case, the server must be available to run the application.

When a user "runs" the deployment manifest, Longhorn downloads and installs the application on the local computer. The application can install icons on the desktop, add Start menu items, and generally become a traditional installed application. Of course, you also get the automatic background updates, version rollback, and uninstall support.

When you first launch a deployment manifest, Longhorn installs the application in the application cache and adds an entry to the Control Panel's Add or Remove Programs list. Subsequently, whenever you run the deployment manifest, the application loads directly from the application cache. It is typically not downloaded again.

However, when you uninstall the application from the cache using the Control Panel's Add/Remove Programs applet, subsequently execution of the deployment manifest downloads and installs the application once again.

Alternatively, you can change the version number of the application on the server. Then, when you run the deployment manifest, Longhorn will download and install the new version side-by-side with the current version. Both versions of the application will appear in the Add or Remove Programs list.

Why Create Yet Another Build System?

I really like MSBuild, even though, at the time of this writing, I've had only a few weeks of experience with it. Of course, years of experience with makefiles makes any more elegant build system attractive. At present, there are two alternative build systems in common use—Make and Ant. It seems natural to compare MSBuild to such alternatives.

Why Not Use Make?

Why develop a new build system when many developers are familiar with an existing one called Make? Make has poor integration of tools into the build system. Make simply executes shell commands. Because of this, there's no inherent ability for one tool to communicate with another tool during the build process. MSBuild creates instances of the *Task* classes, and tasks can communicate among themselves passing normal .NET types.

Makefiles have an unusual syntax, are difficult to write, and don't scale well, as they get complex for large projects. In addition, tools other than Make cannot easily process a makefile. Tools other than MSBuild can easily generate and parse the XML-based syntax of an MSBuild project.

Finally, Make doesn't really have support for projects. There's no file system abstraction, and no support for cascading properties. Moreover, there's no design-time support for generating a makefile.

Why Not Use Ant?

A similar frequently asked question is why develop a new XML-based build system when there's an existing very successful and rich system called Ant? Ant is a Java, open source build system from *Apache.org* that pioneered XML-based project files and tasks as the atomic build operation. There's also a great .NET port of Ant called NAnt available from *nant.sourceforge.net*. On the surface, MSBuild and Ant/NAnt are similar. Both tools use XML as their project serialization format, and both tools use tasks as their atomic unit of build operation. Both tools have their strengths, but when you take a closer look they have different design goals.

Ant made the design decision to place much functionality into a large set of tasks. MSBuild has a different design goal, where similar functionality is encapsulated by the engine (such as timestamp analysis, intertask communication via items, task batching, item transforms, and so on). Both approaches have their strengths and weaknesses.

Ant's model allows developers to extend and control every detail of the build, and therefore it's very flexible. Nevertheless, it also puts a greater responsibility on task writers because tasks need to be much more sophisticated to provide consistent functionality. MSBuild's model lessens the amount of functionality that each task needs to implement. Project authors can therefore rely on consistent functionality across different projects, targets and tasks. In addition, integrated development environments such as Visual Studio can also rely on those services to provide consistent results and a rich user experience, without having to know anything about the internals of the tasks called during the build process.

Similarly, while Ant has the concept of a build script, it does not have the concept of a project manifest that MSBuild has. A build script says how to create a set of files but doesn't provide additional semantics describing how the files are used. A manifest additionally describes the semantics of the files, which allows additional tools, such as an IDE, to integrate more deeply with the build system. Conversely, the lack of a project manifest means a developer can more easily tailor Ant to build new kinds of "stuff" because there's no constraining schema for the build script.

Summary

You've now mastered the basics. You can write XAML and can compile, deploy, and run the resulting application. Unfortunately, the applications you've learned to write so far are pretty boring. Chapter 3 dives into XAML in depth and shows you how to use a wide variety of UI objects provided by the Longhorn platform. Later chapters show you a number of the other new technologies that you can also use in your applications.