Protobuild Documentation

Release latest

June Rhodes

Aug 24, 2017

General Information

Protobuild is a project generation system for C#. It aims to make cross-platform development in C# easier, by generating your projects in the appropriate format for each platform.

Protobuild is fully open source and available under an MIT license.

This documentation is organised into two sections:

- *User Documentation*
- *End User Guides*

CHAPTER 1

User Documentation

This documentation is for developers interested in using Protobuild in their own projects.

Frequently Asked Questions

Why was Protobuild created?

I created Protobuild in July 2013 when I started having to manage targeting 3 desktop platforms for a game. At the time I had 3 project files for every library and executable, and it was no longer maintainable between keeping files and references in sync. I investigated project generators, but the only project generators that were developed were one-way; if you wanted to add new files to your project, you had to manually modify project definitions.

So I set out to build a project generator that would operate in a way that would allow for two-way synchronisation, provide very simple mechanisms for generating projects. I didn't want people to have to install anything either, hence the reason Protobuild ships as a single executable, included in repositories that use it for generation.

In March 2014, Protobuild was adopted by MonoGame. Protobuild is now capable of generating projects for 12 different platforms, and is used by a variety of open and closed source projects.

What platforms can Protobuild generate projects for?

Protobuild supports the following platforms out-of-the-box:

- Android (via Xamarin)
- iOS (via Xamarin)
- Linux
- MacOS
- Ouya (via Xamarin)
- PCL (for Bait-and-Switch PCL only)

- PSMobile
- Windows
- Windows8
- WindowsPhone
- WindowsPhone81
- Web (via JSIL)

Can I extend it to support other platforms?

You can extend Protobuild to support other platforms by modifying the project generation templates. Refer to *Customizing Protobuild* for more information on how to do this.

What license is Protobuild under?

Protobuild is made available under an MIT license.

Why not use MSBuild conditionals to target multiple platforms?

Unfortunately Xamarin Studio (the IDE used to develop Mac OS, Linux, iOS and Android applications) doesn't understand conditionals in C# projects, and likely never will. Since this IDE is used for the majority of cross-platform C# development, it rules out using MSBuild conditionals to solve this problem.

What languages can Protobuild generate for?

C# is currently the only supported platform type, but support for generating C++ projects is planned.

Why does Protobuild have it's own package management system?

NuGet was developed primarily with the focus of packaging .NET libraries, and it uses framework versions to distinguish between different .NET binaries. Unfortunately, this abstraction does not work when libraries are dependent on specific operating system or device APIs (particular those that are P/Invoke'd).

Up until October 2014, the only way to include external libraries in Protobuild was via the submodule mechanism, where you would add a submodule using `git submodule add`, and Protobuild would recursively load all of the project definitions from the submodules.

This works well from the perspective of being able to debug and correct issues in libraries you are using, but suffers problems when the number of libraries, or the amount of history libraries have is large. This is because when you clone your project, you also need to clone the history of every submodule and all of it's dependencies. In addition, you also need to build all of the libraries you are using, even if you rarely or never change them.

To solve the issues of depending on large, cross-platform libraries, while still maintaining the flexibility of debugging libraries when needed, package management was added to Protobuild. This allows libraries to be referenced in either binary or source form transparently, and allows developers to switch between the two formats as needed. Being developed with cross-platform support in mind, it explicitly supports binaries per-platform, rather than per-framework, which allows libraries dependent on platform-specific functionality to be shipped in a package.

What files should I keep in source control?

When using Protobuild, you should place the following files in source control:

- Build\Projects*.definition
- Build\Module.xml
- Protobuild.exe
- Any other source code files

You should not place any of these files in source control, unless they are part of a project not generated by Protobuild:

- Any C# project files
- Any `.sln` solution files

Why does Protobuild use XML for it's project definitions?

Unlike MSBuild, Protobuild project definitions are purely declarative. That is, you can't declare custom tasks or steps to occur during your build inside a project definition file (but you can do this by overriding the project generation template).

This is done to ensure that two-way synchronisation is possible. If Protobuild used a procedural or task based system for declaring projects, then it would be non-trivial to determine how to synchronise changes in C# projects back into the definition files.

I've seen Protobuild projects explicitly set project GUIDs, do I need to do that?

Unless you are migrating an existing library which has users using it in source form, you **do not** need to set any project GUIDs. Protobuild will automatically generate appropriate project GUIDs for all projects you have defined.

If you are migrating an existing library, and want to retain your project GUIDs when migrating to Protobuild, see *Explicitly setting Project GUIDs*.

How can I contribute?

Please refer to the *Contributing* page.

How do I build Protobuild itself?

If you are interested in debugging Protobuild, you should generate projects for your platform and then open the Protobuild solution in your IDE. Set the `Protobuild.Debug` project as the startup project.

If you are interested in contributing or developing a custom version of Protobuild, you'll need to run the `Build/rebuild.sh` script, which will build and compress Protobuild to produce the resulting executable.

Support

There are multiple ways to get support for Protobuild.

GitHub

If you have feature suggestions, or you're running into issues with Protobuild (crashes, bugs or other unexpected behaviour), please file an issue on GitHub.

Twitter

For particularly urgent issues, you can reach out to the primary developer on Twitter. Note that this developer is only active during normal Australian Eastern Standard Time hours (+10 GMT).

Contributing

To contribute to Protobuild, we need you to follow a few important things. Most importantly, make sure you're familiar with two of the core principles of Protobuild:

- **Zero maintainance** - We ensure that you never need to maintain or update your project definitions when new versions of Protobuild are released, because Protobuild ships as a single executable in your repository. There's nothing for you or your users to install in order to generate projects.
- **Guaranteed backwards and forwards compatibility** - When you include a third-party that's also using Protobuild, we guarentee that the version of Protobuild in your repository will be able to invoke the third-party library's copy of Protobuild, even when they're different versions.

These principles place some restrictions on the way that features are developed. Here's are some restrictions to note:

- Any changes to how project GUIDs are generated, loaded or written out is extremely sensitive. We rely on the project GUIDs to be generated in the same way across submodules, and submodules may ship different, older or newer versions of Protobuild.
- The invocation of Protobuild in submodules must be checked using feature queries. We use a command `--query-features` so that Protobuild can find out what command-line features instances of Protobuild have in submodules. This means that if you're changing or adding new command-line arguments that are passed to submodules, you must add a feature check so that newer versions of Protobuild do not invoke older versions of Protobuild incorrectly.

Copyright licensing / agreement

When you submit your PR to GitHub, you will be required to sign the .NET Foundation CLA for Protobuild if you haven't done so already. The automated CLA bot will automatically guide you through the process of signing the CLA if necessary. You can read the CLA ahead of time at the CLA Website.

Check the issue list

We plan feature implementations using the GitHub issue tracker. Before you go and implement something, check the issue tracker to see if there is already a similar feature planned. If there is, you'll need to develop your contribution in alignment with the design already outlined in those feature issues; we can't accept contributions that may act in conflict with a planned design.

We use pull requests

Once you've read and followed the instructions in the rest of this document, you'll need to submit a pull request on GitHub to have your contribution accepted.

You must write functional tests

You **must** write functional tests that cover the new feature you have developed. Pull requests that are missing functional tests won't be accepted or merged.

The Protobuild project currently has at least 71 functional tests that cover the behaviour of the Protobuild executable in various situations. We want to continue extending this functional test suite to cover all of the various invocations of Protobuild in all circumstances to ensure that we do not have regressions in the software.

Run the automated build before submitting

Before your submit your pull request, you must run `Protobuild.exe --automated-build`. This not only produces a compressed executable, but also runs the unit and functional test suites that must pass in order for your contribution to be accepted.

We'll rebase your changes

We **do not** use merges in our Git history. This is because we are storing a binary version of Protobuild in the repository, which also acts as the version of Protobuild that users download.

Once your pull request has been accepted by an owner, that person will clone your change locally, run `Protobuild.exe --automated-build` themselves (to ensure the integrity of the compressed executable), amend the commit, and then rebase and push it onto `master`.

Adhere to the style guide

Protobuild has been organically developed over several years, so you may notice inconsistent naming conventions in the codebase. We intend on bringing these instances in alignment with the style guide, but this is an ongoing process.

Our style conventions are currently:

- Private fields should use camel case with an underscore prefix, e.g. `_somePrivateField`.
- Don't use `this.` prefix unless it's required
- Use sensible C# conventions for class naming, etc.

You must follow the style guide for your contribution to be accepted. We intend on automating style checking at some point in the future, but in the meantime, the person reviewing your code on GitHub may highlight style issues which you will need to fix.

Summary

If you're ready to submit your change for code review, create a pull request on GitHub. Thank you for contributing to Protobuild.

Getting Started

This guide will help you install the graphical tool known as the Protobuild Manager on your computer. The graphical tool provides a friendly interface for opening and creating new projects.

If you prefer working from the command-line, you can create projects using the command-line interface at *Getting Started*.

Download Protobuild

First you'll need to download Protobuild itself:

Install the Protobuild Manager

Double-click the Protobuild executable that you downloaded, or if you're running on Mac OS or Linux, run the following command from the terminal:

```
$ mono Protobuild.exe
```

This will automatically download and install the Protobuild Manager (graphical interface) suitable for your platform.

Run Protobuild Manager

On Windows

If you are on Windows, just double-click Protobuild in a directory that does not contain a project in order to start the Protobuild Manager.

On Mac OS

You can start the Protobuild Manager on Mac OS by running:

```
$ mono Protobuild.exe --execute Protobuild.Manager
```

from a directory that contains `Protobuild.exe`.

On Linux

You can start the Protobuild Manager on Linux by running:

```
$ mono Protobuild.exe --execute Protobuild.Manager
```

from a directory that contains `Protobuild.exe`.

Getting Started

This document will show you how to get up and running with Protobuild in 5 minutes. For this guide we'll assume you're interested in creating a project.

If instead you're looking to migrate an existing project, we recommend following this guide to get acquainted with the way that Protobuild works, and then refer to *Migrating Existing C# Projects* for performing a project migration.

Download Protobuild

First you'll need to download Protobuild itself:

After you have downloaded Protobuild, place it in an empty directory where you want to create your project. Protobuild itself is cross-platform; the same executable will run on all desktop platforms.

Note: If you are using source control, you should add or track Protobuild.exe in your source control system. For more information, refer to *What files should I keep in source control?*

Start with a template

Starting with an existing project template is the easiest way to create a new project in Protobuild.

Open a terminal or command prompt, and navigate to the directory where you placed Protobuild. You can start with a new template by running:

```
$ Protobuild.exe --start http://protobuild.org/commons/Console <name>
```

Replace `<name>` with the name you want to give your new project. This will be the name used when creating the solution file.

If you are running Mac OS X or Linux, you'll need to prefix the above command with `mono` (as with all commands in this documentation that execute Protobuild).

This will create a Protobuild module using the template at the specified URL. You can find more templates by searching the Protobuild package index.

Tip: Common starting templates (including templates for libraries) can be found under the commons organisation on the Protobuild index.

Note: The term `module` means the directory which contains Protobuild.exe, and all of your project definitions collectively. Protobuild modules can have submodules, which are subdirectories that contain their own Protobuild executable and project definitions.

Outside of this Getting Started guide, `project` refers to an individual Protobuild project definition, and not the module as a whole. We just use the term `project` here since it's familiar to readers.

Start from scratch

You can also create an empty Protobuild module if you wish to create project definitions from scratch. To do this, just run the Protobuild executable with no arguments, and it will automatically create the required directories and module configuration.

If you're creating a module from scratch, you'll need to refer to the documentation on the various project types and how to create them:

- *Defining Application Projects*

- *Defining Console Projects*
- *Defining Library Projects*
- *Defining Content Projects*
- *Defining External Projects*

Generating projects

Once you have created your module, you'll need to generate the C# projects so that you can build your code. To generate the C# projects for your module, run the following command:

```
$ Protobuild.exe --generate
```

Tip: By default, this will generate C# projects for the current platform you are running Protobuild on.

You can specify what platform you want to generate C# projects for by passing it as an argument to `--generate`. For example, to generate projects for Windows, use:

```
$ Protobuild.exe --generate Windows
```

You can also synchronise changes you have made in your C# projects back to the project definition files by running:

```
$ Protobuild.exe --resync
```

Tip: This is the default command, and is triggered when double-clicking Protobuild under Windows.

This will synchronise and then regenerate the C# projects. If you don't wish to regenerate the C# projects (to avoid the Visual Studio project reload dialog), you can use the following command instead:

```
$ Protobuild.exe --sync
```

If you want to clean up the C# project and solution files generated by Protobuild, you can run the following command:

```
$ Protobuild.exe --clean
```

Note: All of the above commands accept a platform as an argument. For more detail on what commands are available, try `--help`.

Building code

Once you have generated your C# projects, you can build your code in any of the standard ways supported via .NET, whether that's using Visual Studio, Xamarin Studio, MonoDevelop or `msbuild` / `xbuild` from the command line.

Project generation will have created a solution file in the same directory as Protobuild. You can open this with any of the IDEs to start working on your project.

When adding or removing files in your C# projects from Visual Studio, Xamarin Studio or MonoDevelop, remember to save the projects in the IDE and then run either `--sync` or `--resync` to save your changes back to your definition files.

Further reading

We recommend reading up on the different available project types (linked above in *Start from scratch*). You can also refer to the *Frequently Asked Questions*.

Defining Application Projects

Application projects define a project which will run as a graphical application on an operating system or device. They share a common definition structure with library projects and console projects; only the `Type` attribute on the project definition differs.

Generated projects

Application projects should be used when you want to produce an executable program for a platform or device.

For Windows, Mac OS X and Linux, these projects are generated as Windows Applications (as shown in Visual Studio under the project properties).

If you are on Mac OS X, and have Xamarin Mac installed, we generate the project as a Mac application instead.

For iOS and Android, these projects are generated as Xamarin iOS and Xamarin Android applications respectively. For Ouya, we generate these projects as an Android application.

For Windows8, WindowsPhone and WindowsPhone81 we generate the appropriate project types as specified by Microsoft for these platforms.

For the Web, these projects will have a post-build step that runs JSIL, which in turn produces Javascript and HTML documents to run the application in a web browser.

Project definition location

All project definitions for your module should be placed under the `Build\Module` directory and have a `.definition` extension. The name of the project should match the name of the file as well; for a project called "MyProject", the project definition should reside at `Build\Module\MyProject.definition`.

Tip: This is the location for all project definitions, including application, console, library, content, include and external projects.

Basic structure

The smallest definition for an application project you can have is shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<Project Name="MyApplication" Path="MyApplication" Type="App">
  <References>
    <Reference Include="System" />
    <Reference Include="System.Core" />
  </References>
  <Files>
    <Compile Include="Program.cs" />
  </Files>
</Project>
```

The presence of `Type="App"` ensures this project is an application project.

Already there are some structures which will look familiar to anyone who has viewed the XML of a `.csproj` file. While there are most optional sections than shown in the example above, the two most important (and required) sections are:

Declaring references

The `<References>` section lists the references for this project. These references can be assemblies in the GAC or other Protobuild projects that have been defined (either in this module, or any submodule you have in your repository).

> **Warning:** These are Protobuild references, not .NET references. The only valid tag is `<Reference>`, and the only attribute it accepts is `Include`. To reference external C# projects or binaries, refer to the *Defining External Projects* documentation, and then add a reference to the external project you define.

If you had another project definition for a library, and you wanted your application project to depend on it, you could add a definition like so:

```
<?xml version="1.0" encoding="utf-8"?>
<Project Name="MyApplication" Path="MyApplication" Type="App">
  <References>
    <Reference Include="System" />
    <Reference Include="System.Core" />
    <Reference Include="MyLibrary" />
  </References>
  <Files>
    <Compile Include="Program.cs" />
  </Files>
</Project>
```

Declaring files

The `<Files>` section lists the files that are included in your project. Unlike references, these tags align directly with the types of tags you might see in a `.csproj` file. Indeed, when your project is generated, these tags are copied as-is into the project, if the files should be included for the given platform.

All of the standard C# file tags work here, including `<Compile>`, `<Content>`, `<None>`, `<EmbeddedResource>` and more. They also support files as links through `<Link>` sub-tag.

To restrict what platforms files are included for, you can use the `<Platforms>`, `<IncludePlatforms>` and `<ExcludePlatforms>` sub-tags. For example, if you want to only include an `.icns` file when the project is being generated for Mac OS X, you could use the following definition:

```
<?xml version="1.0" encoding="utf-8"?>
<Project Name="MyApplication" Path="MyApplication" Type="App">
  <References>
    <Reference Include="System" />
    <Reference Include="System.Core" />
  </References>
  <Files>
    <Compile Include="Program.cs" />
    <Content Include="MyApplication.icns">
```

```
      <Platforms>MacOS</Platforms>
    </Content>
  </Files>
</Project>
```

Tip: Within C# code, you can also use `#ifdef` to isolate platform-specific code. Protobuild by default declares `PLATFORM_<name>` when projects are generated; e.g when generating for Windows, `PLATFORM_WINDOWS` will be declared. Refer to the *CustomDefinitions* property for a full list of defaults.

Another example is that you might have functionality which is supported on all platforms, except one or two. A common example might be functionality which relies on the `dynamic` keyword in C#, which is not available on iOS. If you wanted to exclude a file entirely based on the platform, you could use:

```
<?xml version="1.0" encoding="utf-8"?>
<Project Name="MyApplication" Path="MyApplication" Type="App">
  <References>
    <Reference Include="System" />
    <Reference Include="System.Core" />
  </References>
  <Files>
    <Compile Include="Program.cs" />
    <Compile Include="DynamicFunctionality.cs">
      <ExcludePlatforms>iOS</ExcludePlatforms>
    </Compile>
  </Files>
</Project>
```

When you add files through your IDE, and then synchronise the C# projects back to the project definitions, the new files won't have either the `<Platforms>` nor `<ExcludePlatforms>` tags by default. In order to limit new files to one or more platforms, you'll need to run Protobuild with `--sync` to save your changes to the project definitions, then edit the project definition in a text editor to add the `<Platforms>` (or `<ExcludePlatforms>` tag), and then run Protobuild with the `--generate` option to generate the C# projects again (for your IDE).

Limiting platforms

If you have an application project that for one reason or another won't run on certain platforms, you can use the `Platforms` attribute to restrict what platforms this project is generated for.

Commonly this attribute is used when you are defining projects game development tools that produce content, where the tools will only run on desktop platforms.

An example of using the `Platforms` attribute to restrict when a project is generated can be seen below.

```
<?xml version="1.0" encoding="utf-8"?>
<Project Name="MyApplication" Path="MyApplication" Type="App" Platforms="Windows,
↪MacOS,Linux">
  <References>
    <Reference Include="System" />
    <Reference Include="System.Core" />
  </References>
  <Files>
    <Compile Include="Program.cs" />
  </Files>
</Project>
```

Defining Console Projects

Console projects define a project which will run underneath a terminal or command prompt. Protobuild itself is an example of a console application.

Generated projects

Console projects should be used when you don't have a graphical interface to present to the user, or you want to show console output on Windows.

For Windows, Mac OS X and Linux, these projects are generated as Console Applications (as shown in Visual Studio under the project properties).

For iOS and Android, these projects are generated as Xamarin iOS and Xamarin Android applications respectively. For Ouya, we generate these projects as an Android application. Since console applications are unlikely to have the required files to work on these platforms when built, you most likely need to limit the platforms that your console applications are generated for (refer to *Limiting platforms*).

For Windows8, WindowsPhone and WindowsPhone81 we generate the appropriate project types as specified by Microsoft for these platforms. Since console applications are unlikely to have the required files to work on these platforms when built, you most likely need to limit the platforms that your console applications are generated for (refer to *Limiting platforms*).

For the Web, these projects will have a post-build step that runs JSIL, which in turn produces Javascript and HTML documents to run the application in a web browser.

Basic structure

The smallest definition for an console project you can have is shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<Project Name="MyConsole" Path="MyConsole" Type="Console">
  <References>
    <Reference Include="System" />
    <Reference Include="System.Core" />
  </References>
  <Files>
    <Compile Include="Program.cs" />
  </Files>
</Project>
```

The presence of `Type="Console"` ensures this project is an console project.

Project definition location

All project definitions for your module should be placed under the `Build\Module` directory and have a `.definition` extension. The name of the project should match the name of the file as well; for a project called "MyProject", the project definition should reside at `Build\Module\MyProject.definition`.

Tip: This is the location for all project definitions, including application, console, library, content, include and external projects.

Common sections

All of the sections for console projects are identical to those found in application projects. Refer to the following sections under the *Defining Application Projects* documentation:

- *Declaring references*
- *Declaring files*

Defining Library Projects

Library projects define a project which contains common functionality you want to share between multiple projects.

Generated projects

Library projects always generate appropriate projects such that other executable projects on the platform can refer and use them.

The appropriate information to reference library projects on any platform is automatically determined by Protobuild, so you don't need to provide any additional information other than the library name when referencing it.

Basic structure

The smallest definition for an library project you can have is shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<Project Name="MyLibrary" Path="MyLibrary" Type="Library">
  <References>
    <Reference Include="System" />
    <Reference Include="System.Core" />
  </References>
  <Files>
    <Compile Include="MyClass.cs" />
  </Files>
</Project>
```

The presence of `Type="Library"` ensures this project is an library project.

Project definition location

All project definitions for your module should be placed under the `Build\Module` directory and have a `.definition` extension. The name of the project should match the name of the file as well; for a project called "MyProject", the project definition should reside at `Build\Module\MyProject.definition`.

Tip: This is the location for all project definitions, including application, console, library, content, include and external projects.

Common sections

All of the sections for library projects are identical to those found in application projects. Refer to the following sections under the *Defining Application Projects* documentation:

- *Declaring references*
- *Declaring files*

Defining Content Projects

Content projects allow you to include content into your projects in a cross-platform manner. These are frequently used in conjunction with MonoGame-based projects, but are suitable for including content under any system.

Content projects work differently to other types of projects in that there is no `.csproj` produced for them. Instead, projects that reference content projects (in their `<References>` section) will have the files of the content project included with "Copy-on-build" semantics, such that the files are copied into a Content folder on build.

Basic structure

The smallest definition for an content project you can have is shown below.

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentProject Name="MyContent">
  <Source Include="MyContent/" Match="*" />
</ContentProject>
```

When Protobuild loads a content project, it will scan recursively for all of the files that match the expression, under the `Include` folder.

Project definition location

All project definitions for your module should be placed under the `Build\Module` directory and have a `.definition` extension. The name of the project should match the name of the file as well; for a project called "MyContent", the project definition should reside at `Build\Module\MyContent.definition`.

Tip: This is the location for all project definitions, including application, console, library, content and external projects.

File scanning

To understand the semantics of file scanning, we'll use the following example content project:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentProject Name="ExampleContent">
  <Source Include="ExampleContent/" Match="*.png" />
  <Source Include="ExampleContent/" Match="*.jpg" />
  <Source Include="ExampleContent/subfolder/" Match="*.bin" />
</ContentProject>
```

We'll also assume that your module layout looks like the following:

```
- Protobuild.exe
- Build
  - Module.xml
  - Projects
    - MyApplication.definition
    - ExampleContent.definition
- MyApplication
  - Program.cs
- ExampleContent
  - image1.png
  - image2.jpg
  - text1.txt
  - binary1.bin
  - subfolder
    - image3.png
    - image4.jpg
    - text2.txt
    - binary2.bin
```

If you were to generate projects under this configuration, you'd find that the application project might look similar to the following in Visual Studio (or your IDE):

```
- MyApplication (C# project)
  - Content
    - image1.png
    - image2.jpg
    - binary2.bin
    - subfolder
      - image3.png
      - image4.jpg
  - Program.cs
```

All of the files under the Content folder in your project will be set to copy on build and be of the `<Content>` tag type (if you were to view the generated MyApplication `.csproj` file in a text editor).

Note: The `.bin` file was placed at the root of the Content folder because for matching `.bin` files, the include path includes the subfolder.

Referencing content projects

Referencing a content project is done in the same way that any other project is; to reference a content project, add the appropriate `<Reference>` tag as shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<Project Name="MyApplication" Path="MyApplication" Type="App">
  <References>
    <Reference Include="System" />
    <Reference Include="System.Core" />
    <Reference Include="ExampleContent" />
  </References>
  <Files>
    <Compile Include="Program.cs" />
  </Files>
</Project>
```

Setting the primary source directory

If you need your application to know the location where it's content is sourced from, you can add the `Primary="true"` tag to one of the sources declared in your content project.

An example where this might be applicable is when you have a tool for editing game content. The tool itself will be copied to it's output directory (along with a copy of the game content), however when using the tool you'll want to be editing the files from their source.

When this attribute is added, a `.source` file will be created in the include folder, and will be specified as "Copy-on-build" for projects that reference the content project. Thus to find the absolute path to the content, your application should attempt to read the `Content\.source` file (if it exists). This file contains the full, absolute path to the source folder on the current machine, and hence, it should be ignored by your source control system (i.e. you should add `.source` to your `.gitignore` file).

An example of setting a primary source directory is shown below.

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentProject Name="MyContent">
  <Source Include="MyContent/source/" Match="*" Primary="true" />
  <Source Include="MyContent/compiled/" Match="*.bin" />
</ContentProject>
```

Including platform-specific folders

Often content will be compiled for a specific platform. You can use `$(Platform)` in the include path to replace it with the name of the platform being generated for.

An example of including a platform-specific directory is shown below.

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentProject Name="MyContent">
  <Source Include="MyContent/source/" Match="*" />
  <Source Include="MyContent/compiled/$(Platform)/" Match="*.bin" />
</ContentProject>
```

Defining Include Projects

Include projects allow you to specify a set of C# files and / or other resources which should be included in projects that reference the include project. This allows libraries to include additional code or resources into executables that use them.

For example, a library may provide a default main entry point for programs using that library, so that the developer does not have to manually configure application start up (unless they want to). The library can define an include project which is then pulled in by the application so that the main entry point class is defined in the correct (executable) assembly.

Project structure

A simple example of an include project definition is shown below.

```
<?xml version="1.0" encoding="utf-8" ?>
<IncludeProject Name="MyInclude" Path="MyInclude">
  <Files>
    <Compile Include="SomeClass.cs" />
    <Compile Include="AndroidOnly.cs">
      <Platforms>Android</Platforms>
    </Compile>
    <Compile Include="CustomLinkPath.cs">
      <Link>CustomLinkPath.cs</Link>
    </Compile>
  </Files>
</IncludeProject>
```

Project definition location

All project definitions for your module should be placed under the `Build\Module` directory and have a `.definition` extension. The name of the project should match the name of the file as well; for a project called "MyProject", the project definition should reside at `Build\Module\MyProject.definition`.

Tip: This is the location for all project definitions, including application, console, library, content, include and external projects.

Including files

Declaring files for inclusion in an include project uses the exact same structure used in application, console and library projects, including the ability to declare files for specific platforms or services.

See *Declaring files* for documentation on how to declare files in a project.

Referencing include projects

Referencing include projects works like any other Protobuild project, either from an application, console or library project:

```
<?xml version="1.0" encoding="utf-8"?>
<Project Name="MyApplication" Path="MyApplication" Type="App">
  <References>
    <Reference Include="System" />
    <Reference Include="System.Core" />
    <Reference Include="MyInclude" />
  </References>
  <Files>
    <Compile Include="Program.cs" />
  </Files>
</Project>
```

or from an external project:

```
<?xml version="1.0" encoding="utf-8" ?>
<ExternalProject Name="MyExternal">

  <!-- Reference to a Protobuild include project -->
```

```
  <Reference Include="MyInclude" />

</ExternalProject>
```

Apply include projects as extensions

Rather than declare an include project as a reference in a normal project, you can also use the `AppliesTo` attribute on the include project to specify that it should be applied to another project.

This allows you to define include projects that works as extensions - extending the functionality of another project that is not aware of your include project.

You can set the `AppliesTo` attribute on the `IncludeProject` node, like so:

```
<?xml version="1.0" encoding="utf-8" ?>
<IncludeProject Name="MyInclude" Path="MyInclude" AppliesTo="AnotherProject">
  <Files>
    <Compile Include="SomeClass.cs" />
    <Compile Include="AndroidOnly.cs">
      <Platforms>Android</Platforms>
    </Compile>
    <Compile Include="CustomLinkPath.cs">
      <Link>CustomLinkPath.cs</Link>
    </Compile>
  </Files>
</IncludeProject>
```

Limiting platforms

If you have an include project that you only want to apply when certain platforms are being generated for, you can use the `Platforms` attribute to restrict what platforms this include project is included for.

An example of using the `Platforms` attribute to restrict when an include project is included can be seen below.

```
<?xml version="1.0" encoding="utf-8" ?>
<IncludeProject Name="MyInclude" Path="MyInclude" Platforms="Windows,Linux">
  <Files>
    <Compile Include="SomeClass.cs" />
    <Compile Include="AndroidOnly.cs">
      <Platforms>Android</Platforms>
    </Compile>
    <Compile Include="CustomLinkPath.cs">
      <Link>CustomLinkPath.cs</Link>
    </Compile>
  </Files>
</IncludeProject>
```

Defining External Projects

External projects allow you to reference existing C# projects, prebuilt .NET assemblies, native libraries and group multiple project references.

External projects also allow you to toggle these references based on the platform that is being generated.

Project structure

A simple example of an external project definition, which demonstrates all of the available reference types, is shown below.

```
<?xml version="1.0" encoding="utf-8" ?>
<ExternalProject Name="MyExternal">

  <!-- Reference to a GAC assembly -->
  <Reference Include="System.Core" />

  <!-- Reference to another Protobuild project -->
  <Reference Include="MyLibrary" />

  <!-- Reference to a .NET binary on-disk -->
  <Binary
    Name="ExternalLib"
    Path="ThirdParty/ExternalLib.dll" />

  <!-- Reference to native binary on-disk -->
  <NativeBinary Path="ThirdParty/NativeLib.so" />

  <!-- Reference to an external C# project -->
  <Project
    Name="ExternalProj"
    Guid="758CB33D-6EBD-41EA-BB0C-55B1C97A325F"
    Path="ThirdParty/ExternalProj/ExternalProj.csproj" />

</ExternalProject>
```

Project definition location

All project definitions for your module should be placed under the `Build\Module` directory and have a `.definition` extension. The name of the project should match the name of the file as well; for a project called "MyProject", the project definition should reside at `Build\Module\MyProject.definition`.

Tip: This is the location for all project definitions, including application, console, library, content, include and external projects.

Referencing GAC assemblies

You can reference assemblies in the GAC, or assemblies that are otherwise natively provided by the build system, by using the `<Reference>` tag. If the included assembly isn't known as a Protobuild reference, it is assumed to be an assembly provided by the build system.

```
<Reference Include="System.Core" />
```

Referencing Protobuild projects

You can reference other Protobuild projects by using the `<Reference>` tag. This allows you to add multiple references to other Protobuild projects with a single external project, or to make references on other Protobuild projects dependent on the platform.

```
<Reference Include="MyLibrary" />
```

Warning: You can reference other external projects from an external project, but only at the top-level or under a `<Plaform>` tag. You can't reference another external project from within a `<Service>` section.

Referencing .NET assemblies on-disk

You can reference .NET assemblies that are stored within your repository (as binaries), by using the `<Binary>` tag. This tag requires the name of the .NET assembly, and the path to it *from the root of your module*.

```
<Binary
  Name="ExternalLib"
  Path="ThirdParty/ExternalLib.dll" />
```

Referencing native binaries on-disk

You can reference native binaries that are stored within your repository, by using the `<NativeBinary>` tag. This tag requires the path to the native binary, relative *from the root of your module*.

```
<NativeBinary Path="ThirdParty/NativeLib.so" />
```

This will cause the native binary to be added as a file with type None, copy-on-build semantics, and as a link in the root of any project that references this external project. This ensures the native library is copied to the output directory (so that it can be loaded or P/Invoke'd at runtime).

You will almost certainly want to make these references platform specific; refer to *Platform specific references* on how to do this.

Referencing other C# projects

You can reference external C# projects by using the `<Project>` tag. This tag requires the name of the project, it's GUID and the path to it *from the root of your module*.

```
<Project
  Name="ExternalProj"
  Guid="758CB33D-6EBD-41EA-BB0C-55B1C97A325F"
  Path="ThirdParty/ExternalProj/ExternalProj.csproj" />
```

Warning: If the GUID does not exactly match the GUID specified in the `.csproj` file, you will get strange errors when attempting to build your code under MSBuild or Visual Studio.

If you want to add an external C# project to the solution, but not reference it in any project, you can define an external project as shown below, and simply not add a `<Reference>` to it from any Protobuild project:

```
<?xml version="1.0" encoding="utf-8" ?>
<ExternalProject Name="MyExternalNoRef">

  <!-- Reference to an external C# project -->
  <Project
```

```
    Name="ExternalProj"
    Guid="758CB33D-6EBD-41EA-BB0C-55B1C97A325F"
    Path="ThirdParty/ExternalProj/ExternalProj.csproj" />

</ExternalProject>
```

Since Protobuild loads all definitions from the `Build\Projects` folder, the external C# project will still be shown and built in the solution.

Platform specific references

You can make a group of references specific to one platform by using the `<Platform>` tag to surround them. An example of platform specific references is shown below.

```
<?xml version="1.0" encoding="utf-8" ?>
<ExternalProject Name="MyExternal">

  <Reference Include="MyLibrary" />

  <Platform Type="Windows">
    <NativeBinary Path="ThirdParty/Windows/NativeLib.dll" />
    <Project
      Name="ExternalProj"
      Guid="758CB33D-6EBD-41EA-BB0C-55B1C97A325F"
      Path="ThirdParty/ExternalProj/ExternalProj.Windows.csproj" />
  </Platform>

  <Platform Type="MacOS">
    <NativeBinary Path="ThirdParty/MacOS/NativeLib.dylib" />
    <Project
      Name="ExternalProj"
      Guid="758CB33D-6EBD-41EA-BB0C-55B1C97A325F"
      Path="ThirdParty/ExternalProj/ExternalProj.MacOS.csproj" />
  </Platform>

  <Platform Type="Linux">
    <NativeBinary Path="ThirdParty/Linux/NativeLib.so" />
    <Project
      Name="ExternalProj"
      Guid="758CB33D-6EBD-41EA-BB0C-55B1C97A325F"
      Path="ThirdParty/ExternalProj/ExternalProj.Linux.csproj" />
  </Platform>

  <Platform Type="iOS">
    <Reference Include="OpenTK" />
  </Platform>

</ExternalProject>
```

Adding an external reference

Referencing a external project is done in the same way that any other project is; to reference a external project, add the appropriate `<Reference>` tag as shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<Project Name="MyApplication" Path="MyApplication" Type="App">
  <References>
    <Reference Include="System" />
    <Reference Include="System.Core" />
    <Reference Include="MyExternal" />
  </References>
  <Files>
    <Compile Include="Program.cs" />
  </Files>
</Project>
```

Migrating Existing C# Projects

If you have an existing project you wish to convert to Protobuild, you'll need to perform a few steps to convert your current C# projects into project definition files.

If you haven't used Protobuild before, we recommend reading *Getting Started* first so you can get acquainted with how Protobuild works first.

If your project uses #ifdef's

If your project purely uses `#ifdef` to distinguish between platforms, then the migration process is easy.

1. Copy file declarations

For the project you want to convert, you'll need to create a project definition that matches the current type of the project (refer to the relevant documentation pages for the type of project you need).

After this, copy all of the `<Compile>`, `<Content>`, `<EmbeddedResource>`, etc. tags from the C# project file into the definition file, removing the surrounding `<ItemGroup>` tags.

2. Migrate external references

For references to binary or external (unconverted) C# projects, you'll need to create a new definition of an external project, the format of which is described in *Defining External Projects*. Use this to encapsulate external references, and then within your new Protobuild project definitions, reference the external project instead of binaries or C# projects directly.

If your project has multiple C# project files

If you have multiple C# projects, then the transition is going to be reasonably harder. You will need to determine the files present in each platform-specific project and from there, derive appropriate `<Platforms>` tags against the files, as demonstrated in *Declaring files*.

There is an experimental and unsupported shell script that can do this conversion enmass (reading all the project files and determining what declarations intersect). It is called find-platform-dependent-files.sh and can act as a good starting point for the conversion.

Explicitly setting Project GUIDs

By default, Protobuild generates unique GUIDs for all projects, based on their name and the platform being generated.

If you are migrating an existing library to Protobuild, and there are users currently using your library in source form, you'll need to keep the GUIDs the same when you transition to Protobuild, so that existing users of your library won't have the reference in their projects broken when you change.

> **Warning:** If you do specify explicit project GUIDs, Protobuild can't warn you of a GUID conflict, as we use conflicting project GUIDs to detect duplicate external project references in the solution.

You can explicitly set the project GUIDs as per the example below.

```
<?xml version="1.0" encoding="utf-8"?>
<Project Name="MyLibrary" Path="MyLibrary" Type="Library">
  <ProjectGuids>
    <Platform Name="Android">CA9476CF-99BA-4D03-92F2-73D2C5E58883</Platform>
    <Platform Name="Angle">FCF3C8B9-0AAC-4C32-B184-FAE88F6244B5</Platform>
    <Platform Name="iOS">EB8508BB-9849-4CC2-BC0F-8EB5DACB3C47</Platform>
    <Platform Name="Linux">45253CE1-C864-4CD3-8249-4D1319748E8F</Platform>
    <Platform Name="MacOS">46C538E6-C32A-4A8D-A39C-566173D7118E</Platform>
    <Platform Name="Ouya">9BEA875D-66D2-4A5F-B137-83D262A3851D</Platform>
    <Platform Name="PSMobile">FA26E76F-FDFE-4A8C-B5A3-C3B5761E28F9</Platform>
    <Platform Name="Windows">8DE47032-A904-4C29-BD22-2D235E8D91BA</Platform>
    <Platform Name="Windows8">1518563C-ACCA-4A14-8C5D-DDBE93E2605F</Platform>
    <Platform Name="WindowsGL">7D75E618-19CA-4C51-9546-F10965FBC0B8</Platform>
    <Platform Name="WindowsPhone">CAA9A6E4-7690-4DE0-9531-DE0EAEEC9739</Platform>
    <Platform Name="WindowsPhone81">7522B7E9-25A5-4250-B164-42CC2C4ECCAD</Platform>
  </ProjectGuids>
  <References>
    <Reference Include="System" />
    <Reference Include="System.Core" />
  </References>
  <Files>
    <Compile Include="MyClass.cs" />
  </Files>
</Project>
```

Customizing Projects with Properties

Protobuild has many additional properties that you can set on projects to customize how they are generated. You can declare properties in the optional `<Properties>` section of your project, as demonstrated below.

```
<?xml version="1.0" encoding="utf-8"?>
<Project Name="MyApplication" Path="MyApplication" Type="App">
  <Properties>
    <Property Name="RootNamespace" Value="Microsoft.Xna.Framework" />
    <Property Name="RemoveXnaAssembliesOnWP8" Value="false" />
  </Properties>
  <References>
    <Reference Include="System" />
    <Reference Include="System.Core" />
  </References>
  <Files>
```

```
    <Compile Include="Program.cs" />
  </Files>
</Project>
```

A full list of properties is detailed below:

AssemblyName

Applies to: All platforms and project types

Value type: Platform mapping **or** string

Default: The project name

A set of platform mappings that change the assembly name for the project, based on the current platform.

An example of this property in use is:

```
<Properties>
  <AssemblyName>
    <Platform Name="Android">MyProject.Android</Platform>
    <Platform Name="iOS">MyProject.iOS</Platform>
    <Platform Name="Linux">MyProject.Linux</Platform>
    <!-- etc. -->
  </AssemblyName>
</Properties>
```

The following is also valid:

```
<Properties>
  <Property Name="AssemblyName" Value="MyProject.XYZ" />
</Properties>
```

FrameworkVersions

Applies to: All platforms and project types

Value type: Platform mapping

Default: Equivalent to .NET framework 4.0 (Full Profile)

A set of platform mappings that change the framework version and profile for the assembly, based on the current platform.

An example of this property in use is:

```
<Properties>
  <FrameworkVersions>
    <Platform Name="Android">
      <Version>v4.2</Version>
    </Platform>
    <Platform Name="Windows">
      <Version>v4.0</Version>
      <Profile>ClientOnly</Profile>
    </Platform>
    <Platform Name="Linux">
      <Version>v4.5</Version>
    </Platform>
```

```
    <!-- etc. -->
  </FrameworkVersions>
</Properties>
```

PlatformSpecificOutputFolder

Applies to: All platforms and project types

Value type: "True" or "False"

Default: "True"

Determines whether the output paths for binary and object files during compilation should reside in "bin\$(Configuration)" and "obj\$(Configuration)", or whether the platform name should be used in the path (the default). When the platform name is included in the path, the binary folder becomes "bin\$(Platform)\$(Arch)\$(Configuration)" and the object folder becomes "obj\$(Platform)\$(Arch)\$(Configuration)" (where Arch is the .NET concept of platforms, e.g. `AnyCPU`).

Note: In previous versions of Protobuild, this defaulted to False.

ProjectSpecificOutputFolder

Applies to: All platforms and project types

Value type: "True" or "False"

Default: "False"

Determines whether the output paths for binary and object files should include the project name. The option implies `PlatformSpecificOutputFolder`.

When this option is active, the binary and object paths become "bin\$(Project)\$(Platform)\$(Arch)\$(Configuration)" and "obj\$(Project)\$(Platform)\$(Arch)\$(Configuration)" (where Arch is the .NET concept of platforms, e.g. `AnyCPU`).

You only need to enable this option if you are targeting Android and you have more than one project residing in the same directory. Xamarin Studio for Android exhibits a bug where multiple projects having the same object directory will cause issues when linking against native libraries, and this option allows you to work around the issue by ensuring the object folder is unique per project.

CustomDefinitions

Applies to: All platforms and project types

Value type: Platform mapping

Default: See below

A set of platform mappings that change the compile-time constants that are present when compiling C# code.

An example of this property in use is:

```
<Properties>
  <CustomDefinitions>
    <Platform Name="Android">TRACE;ANDROID;GLES;OPENGL</Platform>
```

```
    <Platform Name="iOS">IOS;GLES;OPENGL</Platform>
    <Platform Name="Linux">TRACE;LINUX;OPENGL</Platform>
    <Platform Name="MacOS">MONOMAC;OPENGL</Platform>
    <Platform Name="Ouya">TRACE;ANDROID;GLES;OPENGL;OUYA</Platform>
    <Platform Name="PSMobile">PSM</Platform>
    <Platform Name="Windows">TRACE;WINDOWS;DIRECTX;WINDOWS_MEDIA_SESSION</Platform>
    <Platform Name="Windows8">TRACE;NETFX_CORE;WINRT;WINDOWS_STOREAPP;DIRECTX;
↪DIRECTX11_1;WINDOWS_MEDIA_ENGINE</Platform>
    <Platform Name="WindowsGL">TRACE;WINDOWS;OPENGL</Platform>
    <Platform Name="WindowsPhone">TRACE;SILVERLIGHT;WINDOWS_PHONE;WINRT;DIRECTX</
↪Platform>
  </CustomDefinitions>
</Properties>
```

The defaults for this option are equivalent to:

```
<Properties>
  <CustomDefinitions>
    <Platform Name="Android">PLATFORM_ANDROID</Platform>
    <Platform Name="iOS">PLATFORM_IOS</Platform>
    <Platform Name="Linux">PLATFORM_LINUX</Platform>
    <Platform Name="MacOS">PLATFORM_MACOS</Platform>
    <Platform Name="Ouya">PLATFORM_OUYA</Platform>
    <Platform Name="PSMobile">PLATFORM_PSMOBILE</Platform>
    <Platform Name="Windows">PLATFORM_WINDOWS</Platform>
    <Platform Name="Windows8">PLATFORM_WINDOWS8</Platform>
    <Platform Name="WindowsGL">PLATFORM_WINDOWSGL</Platform>
    <Platform Name="WindowsPhone">PLATFORM_WINDOWSPHONE</Platform>
  </CustomDefinitions>
</Properties>
```

The `DEBUG` constant is always added when the project is a Debug build.

ForceArchitecture

Applies to: All platforms and project types

Value type: String

Default: Any CPU for desktop, varies for mobile

Changes the specified architecture for a project to the given value, even when the project's platform is exposed as Any CPU.

This does not change the C# target platform from Any CPU when exposed at a solution configuration level, it simply forces the architecture of the resulting DLL.

This can be used when you need an assembly to execute in 32-bit mode due to native dependencies.

RootNamespace

Applies to: All platforms and project types

Value type: String

Default: The project name

Changes the root namespace for the assembly.

NoWarn

Applies to: All platforms and project types

Value type: String

Default: None

A comma separated list of compiler warning numbers to disable when compiling the project.

```
<Properties>
  <NoWarn>1591,0436</NoWarn>
</Properties>
```

WarningLevel

Applies to: All platforms and project types

Value type: String

Default: 4

A numeric level indicating the warnings that will be emitted when compiling.

```
<Properties>
  <WarningLevel>0</WarningLevel>
</Properties>
```

TreatWarningsAsErrors

Applies to: All platforms and project types

Value type: "True" or "False"

Default: "False"

Whether or not to treat all warnings as errors.

```
<Properties>
  <TreatWarningsAsErrors>True</TreatWarningsAsErrors>
</Properties>
```

CheckForOverflowUnderflow

Applies to: All platforms and project types

Value type: "True" or "False"

Default: "False"

Enables checked arithmetic in C#. Checked arithmetic causes .NET to throw exceptions when an overflow or underflow occurs, however this results in a performance penalty for the additional checks required.

Note: In previous versions of Protobuild, this was set to true for the iOS platform when built in Debug mode. It is now always off by default, unless enabled by this property.

ManifestPrefix

Applies to: Android and Silverlight platforms

Value type: String

Default: Empty string

Specifies a prefix to be applied to the `Properties\AndroidManifest.xml` and `Properties\AppManifest.xml` paths used by the Android and Silverlight platforms.

RazorGeneratorTargetsPath

Applies to: Only website projects

Value type: String

Default: No effect

Sets the path to the RazorGenerator `.targets` file. When this is specified the `PrecompileRazorFiles` target is called before the build executes.

RemoveXnaAssembliesOnWP8

Applies to: Only Windows Phone 8

Value type: "True" or "False"

Default: "False"

Determines if XNA assemblies should be removed on Windows Phone 8. Set this value to "True" if you are building a MonoGame-based project for Windows Phone 8.

WindowsApplicationIcon

Applies to: Windows platform

Value type: String

Default: Not set

Specify the `.ico` file to use when compiling the application. This sets the icon for the resulting executable.

This option only affects binaries when they are viewed on Windows. Mac OS icons are controlled `separately` by the Info.plist file, and Linux does not directly associate icons with binaries.

iOSUseArmv7

Applies to: Only iOS

Value type: "True" or "False"

Default: "False"

Sets the `MtouchUseArmv7` property. Changing this property may result in incompatibility with included libraries.

iOSUseLlvm

Applies to: Only iOS

Value type: "True" or "False"

Sets the `MtouchUseLlvm` property. Changing this property may result in incompatibility with included libraries.

iOSUseSGen

Applies to: Only iOS

Value type: "True" or "False"

Sets the `MtouchUseSGen` property. Changing this property may result in incompatibility with included libraries.

iOSUseRefCounting

Applies to: Only iOS

Value type: "True" or "False"

Sets the `MtouchUseRefCounting` property. Changing this property may result in incompatibility with included libraries.

iOSI18n

Applies to: Only iOS

Value type: "True" or "False"

Sets the `MtouchI18n` property. Changing this property may result in incompatibility with included libraries.

iOSArch

Applies to: Only iOS

Value type: Comma-separated list of architectures.

Sets the `MtouchArch` property. Changing this property may result in incompatibility with included libraries.

SignAssembly

Applies to: Only iOS

Value type: "True" or "False"

Sets the `SignAssembly` property, which presumably causes the assembly to be signed.

IncludeMonoRuntimeOnMac

Applies to: Only Mac OS with Xamarin Mac

Value type: "True" or "False"

Default: "False"

Sets whether or not the Mono runtime should be included when building on Mac. If the current Mac does not have Xamarin Mac installed, then the resulting project always has this option turned off, regardless of this setting.

MonoMacRuntimeLinkMode

Applies to: Only Mac OS with Xamarin Mac

Value type: "", "SdkOnly" or "Full"

Default: ""

Sets the type of linking mode when the Mono runtime is included on a Mac. If the current Mac does not have Xamarin Mac installed, then the resulting project always has this option turned off, regardless of this setting.

MonoDevelopPoliciesFile

Applies to: All platforms and project types

Value type: String

Default: No effect

Reads the specified file (relative to the module root) and places the contents of this file under the MonoDevelop policies node.

The file contents should be that of applied policies; **not** a policy file exported from MonoDevelop (the formats are different). To retrieve a set of applied policies, you must create a blank C# project in MonoDevelop, apply your policy file and then copy the <*Policies*> node into a seperate file.

> **Danger:** The applied policies file is **not** an XML file. Do not place the XML header (`<?xml ...?>`) in this file as it will be copied literally into the C# project.

When this is complete, the contents of the file should look similar to:

```
<Policies>
  <TextStylePolicy EolMarker="Unix" inheritsSet="VisualStudio" inheritsScope="text/
↪plain" scope="text/plain" />
  <CSharpFormattingPolicy IndentSwitchBody="True" AnonymousMethodBraceStyle="NextLine
↪" PropertyBraceStyle="NextLine" PropertyGetBraceStyle="NextLine"␣
↪PropertySetBraceStyle="NextLine" EventBraceStyle="NextLine" EventAddBraceStyle=
↪"NextLine" EventRemoveBraceStyle="NextLine" StatementBraceStyle="NextLine"␣
↪ArrayInitializerBraceStyle="NextLine" BeforeMethodDeclarationParentheses="False"␣
↪BeforeMethodCallParentheses="False" BeforeConstructorDeclarationParentheses="False"␣
↪BeforeDelegateDeclarationParentheses="False" NewParentheses="False"␣
↪SpacesBeforeBrackets="False" inheritsSet="Mono" inheritsScope="text/x-csharp" scope=
↪"text/x-csharp" />
  <DotNetNamingPolicy DirectoryNamespaceAssociation="PrefixedHierarchical"␣
↪ResourceNamePolicy="FileName" />
  <TextStylePolicy EolMarker="Unix" inheritsSet="VisualStudio" inheritsScope="text/
↪plain" scope="application/xml" />
```

```
  <XmlFormattingPolicy inheritsSet="Mono" inheritsScope="application/xml" scope=
↪"application/xml" />
  <TextStylePolicy EolMarker="Unix" inheritsSet="VisualStudio" inheritsScope="text/
↪plain" scope="text/x-csharp" />
  <StandardHeader Text="This is my file header" IncludeInNewFiles="True" />
  <NameConventionPolicy>
    <Rules>
      <NamingRule Name="Namespaces" AffectedEntity="Namespace" VisibilityMask=
↪"VisibilityMask" NamingStyle="PascalCase" IncludeInstanceMembers="True"␣
↪IncludeStaticEntities="True" />
      <NamingRule Name="Types" AffectedEntity="Class, Struct, Enum, Delegate"␣
↪VisibilityMask="VisibilityMask" NamingStyle="PascalCase" IncludeInstanceMembers=
↪"True" IncludeStaticEntities="True" />
      <NamingRule Name="Interfaces" AffectedEntity="Interface" VisibilityMask=
↪"VisibilityMask" NamingStyle="PascalCase" IncludeInstanceMembers="True"␣
↪IncludeStaticEntities="True">
        <RequiredPrefixes>
          <String>I</String>
        </RequiredPrefixes>
      </NamingRule>
      <NamingRule Name="Attributes" AffectedEntity="CustomAttributes" VisibilityMask=
↪"VisibilityMask" NamingStyle="PascalCase" IncludeInstanceMembers="True"␣
↪IncludeStaticEntities="True">
        <RequiredSuffixes>
          <String>Attribute</String>
        </RequiredSuffixes>
      </NamingRule>
      <NamingRule Name="Event Arguments" AffectedEntity="CustomEventArgs"␣
↪VisibilityMask="VisibilityMask" NamingStyle="PascalCase" IncludeInstanceMembers=
↪"True" IncludeStaticEntities="True">
        <RequiredSuffixes>
          <String>EventArgs</String>
        </RequiredSuffixes>
      </NamingRule>
      <NamingRule Name="Exceptions" AffectedEntity="CustomExceptions" VisibilityMask=
↪"VisibilityMask" NamingStyle="PascalCase" IncludeInstanceMembers="True"␣
↪IncludeStaticEntities="True">
        <RequiredSuffixes>
          <String>Exception</String>
        </RequiredSuffixes>
      </NamingRule>
      <NamingRule Name="Methods" AffectedEntity="Methods" VisibilityMask=
↪"VisibilityMask" NamingStyle="PascalCase" IncludeInstanceMembers="True"␣
↪IncludeStaticEntities="True" />
      <NamingRule Name="Static Readonly Fields" AffectedEntity="ReadonlyField"␣
↪VisibilityMask="Internal, Protected, Public" NamingStyle="PascalCase"␣
↪IncludeInstanceMembers="False" IncludeStaticEntities="True" />
      <NamingRule Name="Fields (Non Private)" AffectedEntity="Field" VisibilityMask=
↪"Internal, Protected, Public" NamingStyle="PascalCase" IncludeInstanceMembers="True
↪" IncludeStaticEntities="True" />
      <NamingRule Name="ReadOnly Fields (Non Private)" AffectedEntity="ReadonlyField"␣
↪VisibilityMask="Internal, Protected, Public" NamingStyle="PascalCase"␣
↪IncludeInstanceMembers="True" IncludeStaticEntities="False" />
      <NamingRule Name="Fields (Private)" AffectedEntity="Field, ReadonlyField"␣
↪VisibilityMask="Private" NamingStyle="PascalCase" IncludeInstanceMembers="True"␣
↪IncludeStaticEntities="False">
        <RequiredPrefixes>
          <String>m_</String>
```

```
        </RequiredPrefixes>
      </NamingRule>
      <NamingRule Name="Static Fields (Private)" AffectedEntity="Field"␣
↪VisibilityMask="Private" NamingStyle="PascalCase" IncludeInstanceMembers="False"␣
↪IncludeStaticEntities="True">
        <RequiredPrefixes>
          <String>m_</String>
        </RequiredPrefixes>
      </NamingRule>
      <NamingRule Name="ReadOnly Fields (Private)" AffectedEntity="ReadonlyField"␣
↪VisibilityMask="Private" NamingStyle="PascalCase" IncludeInstanceMembers="True"␣
↪IncludeStaticEntities="False">
        <RequiredPrefixes>
          <String>m_</String>
        </RequiredPrefixes>
      </NamingRule>
      <NamingRule Name="Constant Fields" AffectedEntity="ConstantField"␣
↪VisibilityMask="VisibilityMask" NamingStyle="PascalCase" IncludeInstanceMembers=
↪"True" IncludeStaticEntities="True" />
      <NamingRule Name="Properties" AffectedEntity="Property" VisibilityMask=
↪"VisibilityMask" NamingStyle="PascalCase" IncludeInstanceMembers="True"␣
↪IncludeStaticEntities="True" />
      <NamingRule Name="Events" AffectedEntity="Event" VisibilityMask="VisibilityMask
↪" NamingStyle="PascalCase" IncludeInstanceMembers="True" IncludeStaticEntities="True
↪" />
      <NamingRule Name="Enum Members" AffectedEntity="EnumMember" VisibilityMask=
↪"VisibilityMask" NamingStyle="PascalCase" IncludeInstanceMembers="True"␣
↪IncludeStaticEntities="True" />
      <NamingRule Name="Parameters" AffectedEntity="Parameter" VisibilityMask=
↪"VisibilityMask" NamingStyle="CamelCase" IncludeInstanceMembers="True"␣
↪IncludeStaticEntities="True" />
      <NamingRule Name="Type Parameters" AffectedEntity="TypeParameter"␣
↪VisibilityMask="VisibilityMask" NamingStyle="PascalCase" IncludeInstanceMembers=
↪"True" IncludeStaticEntities="True">
        <RequiredPrefixes>
          <String>T</String>
        </RequiredPrefixes>
      </NamingRule>
    </Rules>
  </NameConventionPolicy>
</Policies>
```

Changing Module Properties

You can change the default behaviour of Protobuild within your module by modifying the `Build\Module.xml` file. This file contains a number of settings which control Protobuild's default behaviour, and what operations are permitted. These settings do not change how projects are generated.

An example of the MonoGame module configuration is:

```
<?xml version="1.0" encoding="utf-8"?>
<ModuleInfo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
↪www.w3.org/2001/XMLSchema">
  <Name>MonoGame.Framework</Name>
  <ModuleAssemblies />
  <DefaultAction>Generate</DefaultAction>
```

```
  <DefaultWindowsPlatforms>Android,Linux,Ouya,PSMobile,Windows8,Windows,WindowsGL,
↪WindowsPhone,iOS</DefaultWindowsPlatforms>
  <DefaultMacOSPlatforms>MacOS,iOS,WindowsGL,Android</DefaultMacOSPlatforms>
  <DefaultLinuxPlatforms>Linux,WindowsGL</DefaultLinuxPlatforms>
</ModuleInfo>
```

The following module properties are available:

DefaultAction

This can be either "Resync" (default), "Generate" or "Sync". This changes the mode that Protobuild runs in when it is executed with no arguments.

DefaultWindowsPlatforms

When Windows is the current executing platform, and no arguments have been given on the command line, then Protobuild will generate for all of the specified platforms (synchronising with the Windows projects).

DefaultMacOSPlatforms

When Mac OS is the current executing platform, and no arguments have been given on the command line, then Protobuild will generate for all of the specified platforms (synchronising with the Mac OS projects).

DefaultLinuxPlatforms

When Linux is the current executing platform, and no arguments have been given on the command line, then Protobuild will generate for all of the specified platforms (synchronising with the Linux projects).

SupportedPlatforms

A comma-seperated list of supported platform names. If this is not specified, then users can generate for any platform they want (any input string). Known platforms are checked for casing and normalized as appropriate.

If this is specified, then the user will only be able to supply platform names in this list (case insensitive). The specified platforms are checked on input for casing and normalized as appropriate (so specifying "Windows,Linux" in this list will cause an input of "linux" or "LiNuX" to be normalized to "Linux").

DefaultStartupProject

Defines the default startup project for the solution. For example, if you have a project in your module called "ConsoleA", and you want it to be the default for people opening your solution for the first type, specify `<DefaultStartupProject>ConsoleA</DefaultStartupProject>` in Module.xml.

GenerateNuGetRepositories

If this option is "True", then Protobuild will automatically generate a `repositories.config` file based on the `packages.config` files in the repository.

If you have this option enabled, you should exclude `repositories.config` from source control.

DisableSynchronisation

If this option is "True", Protobuild will never synchronise C# projects back to the project definition files. There is almost never any reason to enable this functionality, other than for stylistic reasons.

Including Submodules

When generating projects, Protobuild loads all of the project definitions recursively. That is, it not only loads project definitions from your module's `Build\Projects` folder, but also projects from any submodules that are present.

Note: It is recommended that you use *Package Management with Protobuild* instead of this process.

Adding a submodule

For example, if you have the following project structure that is version controlled by Git:

```
* Build
  * Projects
    * MyProject.definition
  * Module.xml
* MyProject
  * Program.cs
* Protobuild.exe
```

If you want to add MonoGame as a submodule so that you can reference the MonoGame framework, all you need to do is (in the root of your module):

```
$ git submodule add https://github.com/mono/MonoGame.git MonoGame
```

After this command runs you'll end up with:

```
* Build
  * Projects
    * MyProject.definition
  * Module.xml
* MonoGame
  * ...
* MyProject
  * Program.cs
* Protobuild.exe
```

You can now reference MonoGame by adding:

```
<Reference Include="MonoGame.Framework" />
```

in your project definition.

Recursive behaviour

When you run Protobuild in your module, it will be recursively executed in submodules with the same arguments. That is in the example above, if you run "Protobuild.exe –generate Windows" in your module, then "Protobuild.exe –generate Windows" will be called in the MonoGame submodule (and so-on so-forth for it's submodules).

This ensures that all the project files are available when resyncing or generating your module.

Package Management with Protobuild

Protobuild provides a powerful, cross-platform package management system. It is intended to fix some of the shortcomings of NuGet, while also providing greater flexibility for developers.

Primarily the features of Protobuild's package management system over NuGet are:

- Full support for cross-platform projects, in particular those that need to make native calls (such as P/Invoke).
- Automatic fallback to source code repositories if binary versions are not available for the platform.
- The ability to swap source and binary versions of packages, reducing the work required to diagnose issues in a package's functionality.
- Packages are LZMA compressed and deduplicated to significantly reduce their size.
- Support for module templates as packages.

Finding packages

The primary package repository is located on the Protobuild website. This is known as the Protobuild Package Index.

You can use the search functionality on the Protobuild Package Index to find packages to install. The Protobuild Package Index also instructs you on what command to run to add packages.

Adding packages

You can add a package with the following command:

```
$ Protobuild.exe --add <URI>
```

For example, to add the MonoGame package to your module, you can use the following command:

```
$ Protobuild.exe --add http://protobuild.org/MonoGame/MonoGame
```

Tip: You'll notice that when adding MonoGame that Protobuild will clone the MonoGame Git repository. This is because (as of the time of writing), MonoGame does not ship binary Protobuild packages.

Packages can ship binary or source versions, or both. By default, when a binary version is not available for the desired platform, Protobuild will automatically fall back to cloning the source repository (if the package has one).

Note: Protobuild will automatically add the folder containing the package to your `.gitignore` file, if your module is being tracked in a Git repository.

If you want to add a package using a specific branch, you can use the following command:

```
$ Protobuild.exe --add http://protobuild.org/MonoGame/MonoGame@develop
```

If you want to add a package using a specific Git commit, you can use:

```
$ Protobuild.exe --add http://protobuild.org/MonoGame/
↪MonoGame@81bdf42b88acd5e859d856bc0f7fa18004ff977e
```

Switching a package to source format

If you have added a package, and it's currently in binary format, you can get Protobuild to switch the package to a source format with:

```
$ Protobuild.exe --swap-to-source <URI>
```

The URI is the URI of the package. If you don't remember it, you can view `Build\Module.xml`, which contains all of the packages added to your module, and their URIs.

Note: This will only work if the package has a Git repository URL configured.

Switching a package to binary format

If you have added a package, and it's currently in source format, you can get Protobuild to switch the package to a binary format with:

```
$ Protobuild.exe --swap-to-binary <URI>
```

The URI is the URI of the package. If you don't remember it, you can view `Build\Module.xml`, which contains all of the packages added to your module, and their URIs.

Note: This will only work if there is a binary version of the package available for the desired commit and platform.

Upgrading a package

If you have added a package which is tracking a branch (the default), then you can upgrade a package to the latest version of that branch by using the `--upgrade` option, like so:

```
$ Protobuild.exe --upgrade <URI>
```

By default packages usually track the `master` branch of a package. If the package is in binary format, and you want to upgrade a specific platform, you can do so with:

```
$ Protobuild.exe --upgrade <URI> <platform>
```

Warning: If the package is in source format, this command will undo any local modifications you have made. It works by deleting the package's folder and replacing it with a newer version.

Note: You will need to run Protobuild with `--generate` to generate projects in the upgraded packages.

Upgrading all packages

You can upgrade all packages in a module like so:

```
$ Protobuild.exe --upgrade-all
```

If you want to upgrade packages for a specific platform, you can do so with:

```
$ Protobuild.exe --upgrade-all <platform>
```

Warning: This command will undo local modifications to any packages that are in source format. It works by deleting all package folders and replacing them with the latest versions.

Note: You will need to run Protobuild with `--generate` to generate projects in the upgraded packages.

Redirecting package URLs

You can redirect package URLs for packages to use alternate or modified versions. You can configure package URL redirections at a user level, or transiently with command arguments.

To redirect a package URL for a single invocation of Protobuild, use the `--redirect` option like so:

```
$ Protobuild.exe --redirect http://source/path http://target/newpath
```

Most often this argument is used in conjunction with either `--generate` or `--upgrade-all`:

```
$ Protobuild.exe --redirect http://source/path http://target/newpath --upgrade-all
```

You can configure this option at a user level, by creating or modifying a file called `protobuild-redirects.txt` which resides in your Application Data directory (`~/.config/protobuild-redirects.txt` for Linux and Mac OS). The format of this file looks like:

```
# This is a comment
http://source/path -> http://target/newpath
```

Package URLs can also be redirected to a local Git repository residing on your machine. This is useful if you have an internal modified version of a library that you want to use across multiple projects.

To redirect to a local Git repository, specify the target URL as `local-git://` followed by the full path to the Git repository:

```
# On Linux / Mac:
http://source/path -> local-git:///home/user/path/to/repo

# On Windows:
http://source/path -> local-git://C:\Users\user\Documents\path\to\repo
```

Removing packages

At the moment, to remove a package you'll need to open `Build\Module.xml` with a text editor, and remove the section for package you want to remove.

For example, in the module shown below, to remove the Protogame package, you'd remove the highlighted lines.

```
<?xml version="1.0" encoding="utf-8"?>
<ModuleInfo xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://
↪www.w3.org/2001/XMLSchema">
  <Name>MyProject</Name>
  <DefaultAction>resync</DefaultAction>
  <GenerateNuGetRepositories>true</GenerateNuGetRepositories>
  <DisableSynchronisation xsi:nil="true" />
  <ModuleAssemblies />
  <Packages>
    <PackageRef>
      <Uri>http://protobuild.org/hach-que/Protogame</Uri>
      <GitRef>master</GitRef>
      <Folder>Protogame</Folder>
    </PackageRef>
  </Packages>
</ModuleInfo>
```

You'll then need to delete the folder that contains the package.

Creating a package

If you want to create and publish a package for your own library on the Protobuild index, refer to the *Creating Packages for Protobuild* documentation.

Using template packages

You can use template packages to quickly set up new modules. This is detailed in the Getting Started guide under *Start with a template*.

Package Management with NuGet

Although we recommend using Protobuild's package management system, there are many package binaries and libraries available through NuGet.

Protobuild supports using NuGet packages with Protobuild projects, with support for package URLs of the form `https-nuget://repository.org/|PackageName@PackageVersion`.

> **Warning:** NuGet does not offer proper seperation of platforms for binaries. You may find that installed NuGet packages work when targeting Windows, but fail when targeting other desktop or mobile platforms. These will most often manifest themselves as runtime failures or crashes (especially when running under Mono where assumed APIs may not be available).
>
> Also be aware that (as of writing), Portable Class Libraries do not work on Mono under Linux. As such, even if the NuGet package provides a PCL version, it may not work on all platforms.
>
> We highly recommend you ensure that the NuGet packages you have selected work on other platforms as early in the development process as possible.

Adding NuGet packages

You can add a NuGet package with the following command:

```
$ Protobuild.exe --add <URI>
```

You'll need to know the repository URL, package name and package version that you want to add. For example, to add the Ninject package from nuget.org at version 3.2.2.0, you would use the following command:

```
$ Protobuild.exe --add "https-nuget://www.nuget.org/api/v2/|Ninject@3.2.2.0"
```

Note: You will need to quote the URL as it contains a pipe character for seperating the URL and package name. This is because on most systems, the pipe character is used to pipe one command into another.

Referencing NuGet packages

With Protobuild's support for NuGet packages, you can reference a NuGet package like any other. In the project definition file which you want to reference the NuGet package in, add it to the `References` section, like so:

```
<?xml version="1.0" encoding="utf-8"?>
<Project Name="MyApplication" Path="MyApplication" Type="App">
  <References>
    <Reference Include="System" />
    <Reference Include="System.Core" />
    <Reference Include="Ninject" />
  </References>
  <Files>
    <Compile Include="Program.cs" />
  </Files>
</Project>
```

The reference name is the same as the package name you provided during `--add`.

Note: NuGet packages imported through this mechanism always have a single external project available for reference, even if the package has multiple DLLs. When a NuGet package contains multiple DLLs, the external project reference will add a reference to all of them.

Re-pushing packages to a Protobuild package repository

If you want to re-push a NuGet package (or any kind of supported binary package) back to a Protobuild repository, you can do so with the `repush` command, like so:

```
$ Protobuild.exe --repush <apikey> <source_url> <dest_url> <git_hash> <platform>
↪<branch>
```

For example, to re-push a copy of Moq from NuGet to the Protobuild index, you would use a command like this:

```
$ Protobuild.exe --repush <apikey> "https-nuget://www.nuget.org/api/v2/|Moq@4.2.1502.
↪911" http://protobuild.org/nuget/Moq hash:4.2.1502.911 Windows 4.2
```

This is useful if you are running your own Protobuild package repository, and want to ensure the packages continue to be available even if the original repository is offline.

Creating Packages for Protobuild

For an overview of the Protobuild package management system, refer to *Package Management with Protobuild*.

When creating a Protobuild package, you can provide your package in source format, binary format or both.

If your library is stored in a publically accessible Git repository, and uses Protobuild, you can provide the Git URL for cloning when creating your package. This will allow users to use any version of your library on any platform you support, without requiring built packages.

If you want to provide built packages to reduce download and build times, or if your library is not managed by Protobuild, you can provide binary packages. These result in a significantly smaller download and almost no build time cost, but you'll need to build the code, package it and push it to the Protobuild index. This is particularly useful if your source repository has a large history, or has large binary files within it.

Creating your package on the index

To create a package on the Protobuild Package Index, you'll need to login and create a new package. Roughly you'll need to follow the steps below:

- Login to the index. The package index uses Google accounts for authentication.
- When prompted, set your username for your account. Although this can be changed later, no redirects will be configured if you change it.
- On your account page, click "New Package". This will be a URL of `http://protobuild.org/<account name>/new`.
- Specify the name of your new package.
- Specify your package type. The package types available are:
  - **Library:** A library consumed by other modules. These are added to a module with `--add`.
  - **Template:** A module template. These are used with `--start` and are outlined under *Template packages*.
- Specify the full URL to your Git repository if applicable. If you are providing a source version of your package, then you need to have this set.
- Write a short description of your package. Descriptions don't support formatting or styling.
- Click Save.

Note: If your package is a native binary that is not built using Protobuild, or if your package otherwise can't be used or referenced directly in source format, you should omit the Git URL and only provide binary versions of your package.

Automatically creating a package file

Protobuild provides an automatic packaging mechanism which can be used to automate the process of packaging your binaries in most cases.

Note: This only applies to Library packages. Template packages need to follow the manual process outlined in the next section.

Because Protobuild uses a declarative syntax, it can determine the output locations and related files from your build, and automatically include them in your package. In addition, it can automatically generate the required external projects in the binary package required for usage.

To create a package file automatically, you should run the following commands:

```
$ Protobuild.exe --generate Windows
$ Protobuild.exe --build Windows
$ Protobuild.exe --pack . MyPackage-Windows.tar.lzma Windows
```

The first argument to pack (the dot) indicates the directory path to the module being packaged, which in the example above is the current directory.

Repeat this process for each platform you want to build and package.

Warning: You must build your code before running `--pack`, or the binaries will not have been built for inclusion in the package.

The automatic packaging mechanism won't work for all projects; in particular, if you have any external C# projects referenced with the `<Project>` tag, you'll receive a warning that the automatic packaging mechanism could not translate it for that package. In this case, refer to the manual section below on how to create and add the required files as additional content to the automatic packaging mechanism's output.

Note: If the automatic packaging mechanism works for you, skip to *Publishing a package file*.

Manually creating a package file

If you are packaging a template, or if the automatic packaging mechanism is unable to handle your project, you can extend or replace the process using a "filter file". A filter file describes what files should be included during packaging, and can be used to provide files in addition to the automatic packaging mechanism, or can be used to construct a package from scratch.

In the next few sections, we cover the package structure, how to define projects, and how to use filter files.

Package structure

The structure for Protobuild packages (for both Library and Template types), is very similar to the normal structure of a Protobuild module, with the exception that you do not need to include the Protobuild executable in your package.

Normally for the binary version of a package, you'll layout the package similar to the following structure:

```
* /
  * Build
    * Projects
      * MyLibrary.definition
    * Module.xml
  * MyLibrary
```

```
    * MyLibrary.dll
    * NativeLibrary.so
```

Note: Refer to *Template packages* for information on filename replacements performed on template packages.

Defining projects

For your Protobuild module, your projects will currently be defined as normal projects. However, since your binary package won't contain source code, you'll need to define external projects that match the files you are providing.

For the example package above, the contents of `MyLibrary.definition` would be:

```
<?xml version="1.0" encoding="utf-8"?>
<ExternalProject Name="MyLibrary">
  <Binary Name="MyLibrary" Path="MyLibrary\MyLibrary.dll" />
  <NativeBinary Path="MyLibrary\NativeLibrary.so" />
</ExternalProject>
```

Refer to the *Defining External Projects* for more comprehensive documentation on specifying external projects.

Specifying files for inclusion

When building your package from source, it's unlikely to resemble the structure of a package. In order to allow the creation of packages without requiring reorganisation of built files, Protobuild packaging uses a text file known as a filter file.

An example filter file is shown below.

```
# Include project definitions for package
include ^Build/Publish/(.*)\.definition$
include ^Build/Module.xml$

# Include binary files for Linux
include ^MyLibrary/bin/Linux/AnyCPU/Debug/(.*)\.(dll|dll\.config|so)$

# Rewrite file locations
rewrite ^Build/Publish/(.*)$ Build/Projects/$1
rewrite ^MyLibrary/bin/Linux/AnyCPU/Debug/(.*)$ MyLibrary/$1
```

The entries in the filter file are executed in sequence, and the purpose of each entry is detailed below. Lines the start with `#` are treated as comments, and blank lines are ignored.

Any instances of the text `%PLATFORM%` are replaced with the name of the platform passed to `--pack`. This allows you to use a common filter file between platforms.

You will most likely have different filter files for each platform you are providing binaries for, as different platforms will require different files to be included.

include:

Specifies that files whose paths match the regular expression should be included in the package.

exclude:

Specifies that any files currently in the package that match the regular expression should be excluded.

rewrite:

Rewrites the paths of files in the package matching the regular expression to the target expression. This effectively allows you to move files around inside the package when creating it, rather than creating the appropriate directory structure before packaging.

autopackage:

Invokes the automatic packaging mechanism. If you need to provide additional external projects (because of warning shown during the automatic packaging mechanism), you should make the first directive in your filter file to be `autopackage`, and then follow on with additional `include` and `rewrite` directives to include the additional binaries and external project definitions required.

The use of this options requires that the directory passed to `--pack` points to a module. If you don't use this option, then the first argument to `-pack` can point to any directory.

Creating the package file

To create the package file ready for upload, you'll need to run the following command:

```
$ Protobuild.exe --pack . MyPackage.tar.lzma Linux Build/Filter.Linux.txt
```

In this example, the current directory `.` will be packaged into a file called `MyPackage.tar.lzma`, using the filter file located at `Build/Filter.Linux.txt`. The platform `Linux` is specified, which is used by both the automatic packaging mechanism (if invoked by `autopackage`), and is also used to replace the text `%PLATFORM%`.

If you want to upload your package via the Protobuild index web interface, you'll need to pack it into `.tar.gz` format instead, as the web interface does not support uploading `.tar.lzma` files (although they provide a much greater compression ratio). You can package as `.tar.gz` using the following command:

```
$ Protobuild.exe --pack . MyPackage.tar.gz Linux Build/Filter.Linux.txt --format tar/
↪gzip
```

Publishing a package file

To publish a package file to the Protobuild index, you'll need to retrieve your API key from your account page (it is listed at the bottom after all of your packages).

With your API key, you can now publish the package file as a new version of your package with the following command:

```
$ Protobuild.exe --push <apikey> MyPackage.tar.lzma http://protobuild.org/MyAccount/
↪MyPackage <githash> <platform>
```

You need to specify a Git hash that matches the hash that the source code was built from. You can retrieve the current Git hash by running:

```
$ git rev-parse HEAD
```

Warning: Package versions are immutable. Once you push a file for a specific Git hash, you can't upload a new version with the same Git hash, unless you delete it from web interface manually.

If you need an identifier that updates as you release new versions, you should update a branch, and direct your users to use the branch as the version.

You can create and update a branch to point at the pushed version by specifying a branch name as the last argument. In the previous example, if you wanted to update the "master" branch to the newly pushed package, you can use:

```
$ Protobuild.exe --push <apikey> MyPackage.tar.lzma http://protobuild.org/MyAccount/
↪MyPackage <githash> <platform> master
```

Template packages

Template packages contain templates for a module. They are structured in the exact same way as library packages, except that when the user creates a new module using a template, the following process happens:

- As files are extracted into the module directory, file and directory names have `{PROJECT_NAME}` replaced with the project name the user gave to `--start`.
- As files are written, `{PROJECT_NAME}` is replaced with the project name, and `{PROJECT_XML_NAME}` is replaced with the project name with characters escaped for XML documents. This process does not apply to binary files.

The Protobuild common templates repository provides a good reference for creating template packages.

Code Signing for iOS

As of writing, Xamarin stores the selected iOS Code Signing key in the C# project. This means that without Protobuild, iOS projects can't be moved across computers without updating that property in the C# project.

When generating C# projects for iOS, Protobuild will check for the existence of a `.codesignkey` file in your home directory, and will use the contents of that file for the property. This allows each computer to have it's own code signing settings, without the project or definition file requiring changes.

Targeting the Web Platform

Protobuild supports targeting C# code for the Web via JSIL, a CIL to Javascript compiler. Protobuild configures projects to use the JSIL compiler as a post-build step transparently, so you don't need to perform any configuration to take advantage of JSIL.

Usage

The JSIL compiler is invoked on application projects.

To generate for the Web platform, use the following command:

```
$ Protobuild.exe --generate Web
```

When your project builds, the JSIL compiler will be invoked and the resulting Javascript files will reside inside the standard bin directory. You can then use the compiled Javascript according to the documentation found on the JSIL wiki.

JSIL Download

The first time you target the Web platform, Protobuild will automatically download and use pre-built JSIL binaries. These binaries will also be used for future projects targeting the Web platform.

You do not need to include JSIL in your project to target the Web platform.

Service Dependencies

Service dependencies allow you to conditionally include submodules and functionality, based on what the consumers of your library require. It can also be used to provide multiple backends for APIs.

Note: Service dependencies are an advanced feature of Protobuild. It is highly recommended that you're well versed in creating Protobuild projects before using service dependencies.

What are services in Protobuild?

Services are declared within projects which are usable by other libraries. They allow you to declare functionality in your library optional, or to provide functionality in multiple ways.

For example, MonoGame on Windows can use one of three graphics backends; it can use DirectX, OpenGL or ANGLE for providing graphics rendering.

In the case of MonoGame, each of these backends is declared as a service, and they are all declared such that they conflict with one another (so you can only have one enabled at a time). While DirectX is the default graphics backend for MonoGame on Windows, it can be explicitly changed when generating projects by running:

```
$ Protobuild.exe --generate Windows --enable MonoGame.Framework/OpenGLGraphics
```

Using services

Continuing the MonoGame example, the OpenGL backend can also be enabled by depending on it from another project (in this example, it would most likely be a MonoGame-based game). This dependency could be represented in the `<Dependencies>` section:

```
<?xml version="1.0" encoding="utf-8"?>
<Project Name="MyApplication" Path="MyApplication" Type="App">
  <References>
    <Reference Include="System" />
    <Reference Include="System.Core" />
  </References>
  <Dependencies>
    <Uses Name="MonoGame.Framework/OpenGLGraphics">
      <Platforms>Windows</Platforms>
    </Uses>
  </Dependencies>
  <Files>
    <Compile Include="Program.cs" />
  </Files>
</Project>
```

For another example, Protogame optionally ships with support for OpenCL and optionally provides a physics engine. By default, these services are not enabled, because it adds build time (more code and more third party libraries need to be built).

In order to enable the physics engine in Protogame, a dependent project would add:

```
<?xml version="1.0" encoding="utf-8"?>
<Project Name="MyApplication" Path="MyApplication" Type="App">
  <References>
    <Reference Include="System" />
    <Reference Include="System.Core" />
  </References>
  <Dependencies>
    <Uses Name="Protogame/Physics" />
  </Dependencies>
  <Files>
    <Compile Include="Program.cs" />
  </Files>
</Project>
```

In the case of Protogame, it uses the Jitter library to provide physics. Service dependencies are smart enough to recognise when a service-aware project in a submodule has no references to it; in this scenario the submodule's project is omitted from the build to reduce build time. Thus if the Physics service is not enabled in Protogame, the Jitter project would not be built.

Note: A service-aware project is one that declares services using the `<Services>` section (see below).

Tip: In the examples above, you'll notice that there's no `<Reference>` to either MonoGame or Protogame. This is because declaring that a project uses a service automatically adds a reference to the project that provides that service.

An alternative to the `<Uses>` tag is the `<Recommends>` tag. Whereas the Uses tag declares a requirement, the Recommends tag enables a service only if it would not conflict with any other enabled service.

Declaring services

You can declare services in your own projects using the optional `<Services>` section as demonstrated in the example below.

```
<?xml version="1.0" encoding="utf-8"?>
<Project Name="MyLibrary" Path="MyLibrary" Type="Library">
  <Properties>
    <CustomDefinitions>
      <Platform Name="Windows">USE_PRIMARY</Platform>
    </CustomDefinitions>
  </Properties>
  <References>
    <Reference Include="System" />
    <Reference Include="System.Core" />
  </References>
  <Dependencies>
    <Uses Name="SecondaryGraphics">
      <Platforms>MacOS,Linux</Platforms>
    </Uses>
    <Recommends Name="PrimaryGraphics">
```

```
      <Platforms>Windows</Platforms>
    </Recommends>
  </Dependencies>
  <Services>
    <Service Name="PrimaryGraphics">
      <Platforms>Windows</Platforms>
      <Conflicts>SecondaryGraphics</Conflicts>
      <Requires>PrimaryAdditional</Requires>
    </Service>
    <Service Name="SecondaryGraphics">
      <RemoveDefine>USE_PRIMARY</RemoveDefine>
      <AddDefine>USE_SECONDARY</RemoveDefine>
      <Platforms>Windows,MacOS,Linux</Platforms>
      <Conflicts>PrimaryGraphics,PrimaryAdditional</Conflicts>
    </Service>
    <Service Name="PrimaryAdditional" />
  </Services>
  <Files>
    <Compile Include="Program.cs" />
  </Files>
</Project>
```

The options available to declared services are:

- **AddDefine:** A comma seperated list of definitions to add to the project, so that code can be enabled / disabled based on the presence of the service.
- **RemoveDefine:** A comma seperated list of definitions to remove from the project, so that code can be enabled / disabled based on the presence of the service.
- **Reference:** Add another Protobuild reference to this project when this service is used.
- **Conflicts:** This service conflicts with another service, and as such they can not both be enabled at the same time.
- **Requires:** This service requires another service to also be enabled, and will cause it to be enabled (if conflicts would occur, an error occurs).
- **InfersReference:** If set to False, then using this service will not imply a .NET reference on the project. This should be set to False when tools or executables are providing services for the purposes of having them built along side the project.

Referencing services

Services can be referenced as their local names (in the context of the previous example, `PrimaryGraphics`), or as their full names; `MyLibrary/PrimaryGraphics`. This local / full name referencing applies anywhere service names are used.

Recommending default services for a platform

You can recommend a default service for a platform, while still giving the user the ability to change to an alternative as demonstrated in the example below.

```
<?xml version="1.0" encoding="utf-8"?>
<Project Name="MyLibrary" Path="MyLibrary" Type="Library">
  <Properties>
    <CustomDefinitions>
```

```
      <Platform Name="Windows">USE_PRIMARY</Platform>
    </CustomDefinitions>
  </Properties>
  <References>
    <Reference Include="System" />
    <Reference Include="System.Core" />
  </References>
  <Dependencies>
    <Uses Name="SecondaryGraphics">
      <Platforms>MacOS,Linux</Platforms>
    </Uses>
    <Recommends Name="PrimaryGraphics">
      <Platforms>Windows</Platforms>
    </Recommends>
  </Dependencies>
  <Services>
    <Service Name="PrimaryGraphics">
      <Platforms>Windows</Platforms>
      <Conflicts>SecondaryGraphics</Conflicts>
      <Requires>PrimaryAdditional</Requires>
    </Service>
    <Service Name="SecondaryGraphics">
      <RemoveDefine>USE_PRIMARY</RemoveDefine>
      <AddDefine>USE_SECONDARY</RemoveDefine>
      <Platforms>Windows,MacOS,Linux</Platforms>
      <Conflicts>PrimaryGraphics,PrimaryAdditional</Conflicts>
    </Service>
    <Service Name="PrimaryAdditional" />
  </Services>
  <Files>
    <Compile Include="Program.cs" />
  </Files>
</Project>
```

In this example, `SecondaryGraphics` must be used on Mac OS X and Linux, but Windows can use either graphics service (and defaults to `PrimaryGraphics`).

Including and excluding files

You can include and exclude files in your project based on the presence of services. This works in a similar manner as platform-based include / exclude, except that the tag names are different.

See the example below for a demonstration of including and excluding files.

```
<Compile Include="MyClass.OpenGL.cs">
  <Services>GLBackend</Services>
</Compile>
<Compile Include="MyClass.DirectX.cs">
  <ExcludeServices>GLBackend</ExcludeServices>
</Compile>
```

Including external references

You can also include and exclude references in external projects based on the presence of services. This works in a similar manner to platform-based inclusion in external projects.

```
<?xml version="1.0" encoding="utf-8" ?>
<ExternalProject Name="MyExternal">

  <Platform Name="Windows">
    <Service Name="PrimaryGraphics">
      <NativeBinary Path="ThirdParty/Windows/Primary.dll" />
    </Platform>

    <Service Name="SecondaryGraphics">
      <NativeBinary Path="ThirdParty/Windows/Secondary.dll" />
    </Platform>
  </Platform>

  <Platform Name="MacOS">
    <Service Name="SecondaryGraphics">
      <NativeBinary Path="ThirdParty/MacOS/Secondary.dylib" />
    </Platform>
  </Platform>

  <Platform Name="Linux">
    <Service Name="SecondaryGraphics">
      <NativeBinary Path="ThirdParty/Linux/Secondary.so" />
    </Platform>
  </Platform>

</ExternalProject>
```

You can also declare services in external projects, which can be used when creating packages that involve service-aware projects.

```
<?xml version="1.0" encoding="utf-8"?>
<ExternalProject Name="MyLibrary">

  <!-- The dependencies and services we declared in our source project -->

  <Dependencies>
    <Uses Name="SecondaryGraphics">
      <Platforms>MacOS,Linux</Platforms>
    </Uses>
    <Recommends Name="PrimaryGraphics">
      <Platforms>Windows</Platforms>
    </Recommends>
  </Dependencies>

  <Services>
    <Service Name="PrimaryGraphics">
      <Platforms>Windows</Platforms>
      <Conflicts>SecondaryGraphics</Conflicts>
      <Requires>PrimaryAdditional</Requires>
    </Service>
    <Service Name="SecondaryGraphics">
      <Platforms>Windows,MacOS,Linux</Platforms>
      <Conflicts>PrimaryGraphics,PrimaryAdditional</Conflicts>
    </Service>
    <Service Name="PrimaryAdditional" />
  </Services>

  <!-- Add the binary reference based on what service is enabled -->
```

```
  <Service Name="PrimaryGraphics">
    <Binary Name="MyLibrary" Path="MyLibrary/MyLibraryWithPrimary.dll" />
  </Platform>

  <Service Name="SecondaryGraphics">
    <Binary Name="MyLibrary" Path="MyLibrary/MyLibraryWithSecondary.dll" />
  </Platform>

</ExternalProject>
```

Customizing Protobuild

You can customize Protobuild's project generation by overriding the project generation templates that it uses.

Warning: Customizing the project generation templates is a very advanced feature of Protobuild. It is highly recommended that you're well versed in creating Protobuild projects before customizing the project templates.

Specifying additional transforms

You can specify additional XSLT transforms to take place without overriding the whole project generation template. This can be used if you need to support additional steps in your build process.

To specify additional XSLT transforms during project generation, create an XSLT file at `Build\AdditionalProjectTransforms.xslt`. An example of this file's contents is shown below (this example allows projects to have protocol buffers precompiled with protobuf-net).

```
<xsl:if test="/Input/Properties/PrecompileProtobuf = 'True'">
  <Target Name="AfterBuild">
    <Exec>
      <xsl:attribute name="WorkingDirectory">
        <xsl:value-of select="/Input/Generation/RootPath" />
      </xsl:attribute>
      <xsl:attribute name="Command">
        <xsl:if test="/Input/Generation/Platform = 'Linux' or /Input/Generation/
↪Platform = 'MacOS' or /Input/Generation/Platform = 'iOS'">
          <xsl:text>mono </xsl:text>
          <xsl:text>Protogame/ThirdParty/protobuf-precompile/precompile.exe </
↪xsl:text>
        </xsl:if>
        <xsl:if test="/Input/Generation/Platform = 'Windows' or /Input/Generation/
↪Platform = 'Android'">
          <xsl:text>Protogame\ThirdParty\protobuf-precompile\precompile.exe </
↪xsl:text>
        </xsl:if>
        <xsl:value-of select="/Input/Properties/PrecompileProtobufInputPath" />
        <xsl:text> -o:</xsl:text>
        <xsl:value-of select="/Input/Properties/PrecompileProtobufOutputPath" />
        <xsl:text> -t:</xsl:text>
        <xsl:value-of select="/Input/Properties/PrecompileProtobufSerializerName" />
        <xsl:text> -access:Public</xsl:text>
        <xsl:if test="/Input/Generation/Platform = 'Linux'">
```

```
          <xsl:text> -f:/usr/lib/mono/4.0/</xsl:text>
        </xsl:if>
        <xsl:if test="/Input/Generation/Platform = 'MacOS' or /Input/Generation/
↪Platform = 'iOS'">
          <xsl:text> -f:/Library/Frameworks/Mono.framework/Versions/Current/lib/mono/
↪4.0/</xsl:text>
        </xsl:if>
        <xsl:if test="/Input/Generation/Platform = 'Android'">
          <xsl:text> -f:MonoAndroid\v1.0</xsl:text>
        </xsl:if>
      </xsl:attribute>
    </Exec>
  </Target>
</xsl:if>
```

This file is copied as-is into the project generation templates before they are used to generate the projects.

Tip: For more information about what is supported inside additional project transforms, extract the full templates as described below. The additional project transforms document has access to all of the functionality that the full generation templates do.

Extracting the XSLT templates

You can instruct Protobuild to extract the templates by running:

```
$ Protobuild.exe --extract-xslt
```

This will place the extracted templates under the `Build\`. When the templates are files in this location, they are used instead of the versions embedded into the Protobuild executable.

Modifying Protobuild

If you need additional functionality within the Protobuild executable, you can build your own version of Protobuild and ship it within your repository.

You can build a modified version of Protobuild by following the instructions in the FAQ: *How do I build Protobuild itself?*

CHAPTER 2

End User Guides

This documentation is for end users; that is, people consuming projects that already use Protobuild. If you're a developer interested in using Protobuild in your own projects, refer to the *User Documentation* instead.

When documenting how to generate your projects, it is recommended you refer your users to the guides listed below.

Usage Guide - Generating Projects

You've been directed here because the project you're viewing uses Protobuild to generate it's projects.

If you are on Windows, you can quickly generating projects for the default platforms by double-clicking on the Protobuild executable in the root of the repository.

If you are on Mac OS X or Linux, you'll need to open a terminal, and once in the root of project repository, run:

```
$ mono Protobuild.exe --generate
```

You'll need Mono installed if you are on Mac OS X or Linux.

If you want to use Protobuild for your own projects, and potentially use this project as a submodule, refer to the *User Documentation*.