Packaging a C# Desktop Application for Debian-based Distros

Barry W. Block
bwb@fireflysoftware.com
29 Aug, 2013
(rev g)

Note: This tutorial is based on the *Introduction to Debian Packaging* which can be found here: https://wiki.debian.org/IntroDebianPackaging#Introduction_to_Debian_Packaging. Many thanks to its author(s) for providing such a straight-forward tutorial that even a noob like myself could follow it. ;-)

Note: This PDF is hosted on Scribd.com.[1] Before working through the tutorial, be sure that you have the latest revision by downloading the .PDF from Scribd.com. This tutorial requires that you copy commands into the Linux terminal. *For this to work properly, you should work from a downloaded .PDF file and not directly from the Scribd web page.*

Introduction

So you've been working tirelessly for months on end, polishing that wonderful C# desktop application of yours and now you're ready to share it with all of those who use Debian-based Linux distros but you haven't a clue how to package it? Not to worry! Packaging applications (including C#/Mono applications) for inclusion into the Debian repositories isn't as difficult as you might think. Having attempted packaging my own C# applications on Linux in the past[2], I was more than pleasantly surprised to find recently that the process has become almost trivial compared to what it once was and I'm certain that others could benefit from the recent success that I've had with packaging my own application.

In this tutorial I'll step you through the process of packaging a C# desktop application from the ground up, starting with nothing more than the application's source. The first step will be to create a "tarball" from the directory that contains the source. Essentially, a "tarball" is nothing more than an archive of the source code and scripts or makefiles that are needed to build a piece of software. It's this archive file that Linux users often build an application from when their Linux distro doesn't yet have the desired application in its repositories. Likewise, it's this same archive file that a Debian packager often starts with when packaging an "upstream" developer's application for inclusion in the Debian repositories.

1 The PDF can be found on Scribd at: http://www.scribd.com/doc/160967740/Packaging-a-C-Desktop-Application-for-Debian-based-Distros.
2 I've been a happy convert to Linux since about 2008.

Here, we will be acting the part of the Debian packager except that instead of starting with a tarball, we'll have to create one ourselves, from the application's source. Overall, the steps in the process of building the Debian (binary) package are as follows:

1. Create a tarball of the source that's being packaged.
2. To the directory from which this tarball was created, add a *debian* subdirectory and populate it with various packaging-related files.
3. Build the package.
4. Test install the package.

Just Some Preliminaries

All of the steps of this tutorial were worked out with my computer running on a Linux live CD (in particular, the Linux Mint 15 live CD, MATE (pronounced "*may-tay*") edition).[3] I highly recommend that anyone working through this tutorial do so while also running on a live CD/USB, virtual PC, a junk computer etc., if only for the peace of mind it may bring in knowing that, no matter what mistakes are made, your system is reasonably safe from harm.[4] Granted, this might not be a precaution that everyone can conveniently take, so whatever works best for you. Just be aware that there could be some risk of damage as a result of installing an improperly created package and that you alone assume all risk of any such damage that might occur to your system if you choose to take no precaution.

I'm going to assume that you'll be working from a live CD/USB and if not, you may need to adjust the instructions a bit here and there. I'll also assume that you know how to connect to the internet once you have booted into the live CD.

As noted at the beginning of this document, *this tutorial should not be followed directly from the Scribd web page* as the copying of commands into the terminal will not work as expected. If the instructions below require that you run a set of commands, (two or more commands) you should be able to select and copy all of them into the Linux terminal in one go. When working directly from the Scribd page this fails to work and you must select/copy a single line at a time which kind of defeats the whole purpose of using the terminal to expedite working through the tutorial in the first place.

Really Getting Started

With all of that out of the way, let's get started. For our packaging work, we'll need to install some development tools. As with other Debian-based distros, in the MATE edition of Linux Mint, the GUI way to do this would be to open up the *Synaptic Package Manager*, or "Package Manager" as it's called

3 You can download the latest Mint "live CD" images at http://www.linuxmint.com/download.php. I like working from a live CD when venturing into unfamiliar locales of the Linux landscape as it provides a convenient means of making mistakes without being brutally punished as a result. Another advantage of exploiting live CDs here is that anyone can follow along in a Linux tutorial regardless of the O/S they're currently running. Whether your desktop's underlying operating system is Linux, OSX or Windows, a live CD doesn't care.

4 If you do decide to boot from a live image, you'll need to use a recent Linux distro release which is still supported so that software packages can be installed onto it.

in the Mint menu. However, for expediency, we'll use the tried and true method of... *the terminal*. Ok, stop yer shakin'—it's not that bad! In fact, the terminal is our friend here. Trust me. ;-)

So, go ahead and open the terminal[5] and run the following command:

```
sudo apt-get update
```

To make running the command easier, you can copy it into your terminal and press the Enter key. Better yet, you can simply *select the above text and drag it straight into the terminal.* Note that depending on whether you selected the trailing end-of-line, you may still need to press the Enter key afterwards so that the command gets executed.

So, what the above command does is update the package cache so that the system has the latest package information available when packages are installed. It should generate quite a bit of text output but unless you see lots of error statements[6], everything should be fine.

Next, we'll begin installing some software packages. Run this command in the terminal:

```
sudo apt-get install build-essential devscripts debhelper gdebi git
```

When asked "Do you want to continue?" just press "y" followed by the Enter key. If the current version of a package is already installed on your system (as will likely be the case for both *gdebi* and *git* if you're running Linux Mint), you'll be told so as shown here:

```
mint@mint ~ $ sudo apt-get install … git
…
git is already the newest version.
…
```

After installing the above packages, the next step is to obtain the application source that we want to package. For this purpose, we're going to download the source for my modest C# desktop application, *PipeWrench* which is hosted on Github.

Note: Although PipeWrench's version might be something other than "1.0" when you work through this tutorial, we'll nonetheless consider its version to be "1.0" for our purposes here. When it comes time for you to actually package your own software for inclusion into the software repositories, you'll want to use whatever version number is current for your software.

In the terminal, run the following commands:

```
cd ~/Desktop
mkdir PipeWrench-Packaging
cd PipeWrench-Packaging
git clone git://github.com/bblock/PipeWrench.git pipewrench-1.0
```

This creates a directory, "~/Desktop/PipeWrench-Packaging/pipewrench-1.0" and then downloads the source for PipeWrench into it. It's from this directory that we'll create our tarball archive. Once archived, this directory (pipewrench-1.0) will become the archive's top-level directory and it must be

5 To do this in Linux Mint/MATE you would press the "Windows" key, type "term", down-arrow once to "Terminal" and press Enter.
6 Did you forget to connect to the internet?

named according to Debian policy. In particular, its name should adhere to this format:

> <package name>-<version>

Here, <package name> is simply “pipewrench” and <version> is the version of the software itself (and again, we'll just assume it to be “1.0”). Typically, a packager working from an existing “upstream” tarball would have to rename this top-level directory after extracting the tarball in order to meet these formatting requirements; however, in this case, I've provided the extra argument to the above “git clone” command so that the PipeWrench source is downloaded into the properly named directory.

The next step in the sequence is to actually create the tarball archive. Debian policy requires that a tarball be named a very specific way. In particular, it should be named as follows:

> <package name>_<version>.orig.<archive type>

Here, both <package name> and <version> are the same as we saw previously. The <archive type> can be one of many different archive file extensions, but we'll be using “.tar.gz” for our tarball's file extension. So, with that in-mind, our tarball will be named, “pipewrench_1.0.orig.tar.gz”. Notice that the hyphen (-) used previously to separate the package name and version in the top-level directory has been replaced in the tarball's name by an underscore (_).

We're now ready to create the tarball, except for one minor problem. Every git repository contains a hidden “.git” directory which we don't want to be included in our Debian package and so we should delete it before proceeding. From the terminal, enter and run the following commands:

```
cd ~/Desktop/PipeWrench-Packaging/pipewrench-1.0
rm -rf .git
```

This should remove the “.git” directory, making the PipeWrench source all clean and proper in preparation to have its picture taken. ;-) Now, from the terminal, run the following commands:

```
cd ~/Desktop/PipeWrench-Packaging
tar -cvzf pipewrench_1.0.orig.tar.gz pipewrench-1.0
```

This results in our much anticipated source tarball, pipewrench*_1.0.orig.tar.gz* being created in the *PipeWrench-Packaging* directory.

Note: It's worth drawing attention to where the tarball is created here because the packaging tools actually expect it to reside at this location relative to the main (top-level) source directory.

Note: Once you've created a tarball from the source, be sure not to edit the source without also editing the corresponding source contained in the tarball—the two must remain in sync or you'll get errors when building the package.

Debianizing the Source

Now that we have our tarball, we're now going to begin “debianizing” the source by adding a *debian* directory beneath the pipewrench-1.0 directory. From the terminal, run the following commands:

```
cd ~/Desktop/PipeWrench-Packaging/pipewrench-1.0
mkdir debian
```

Inside of this new *debian* subdirectory, we'll create the following packaging files:

changelog
compat
control
copyright
install
pipewrench.links
pipewrench.manpages
rules
source/format

Packaging File: changelog

The *changelog* file is a record of all changes that are made to the Debian package. Being as the package we're creating is new, there will be but a single recorded entry in the log indicating the package's initial creation. As Debian packaging tools are rather strict when it comes to the formatting of packaging files overall, care must be taken to properly format the contents of the *changelog* file. Fortunately, there's a tool that can help us to properly create our *changelog* file. That tool is *dch*. From the terminal, run the following commands:

```
cd ~/Desktop/PipeWrench-Packaging/pipewrench-1.0
dch --create -v 1.0-1 --package pipewrench
```

This will create the initial *changelog* file and then present you with a list of editors from which you can choose one to finish editing the file. The default choice is *Nano*, a simple, lightweight terminal based editor that's installed on virtually every Linux distro. Those who are following the tutorial from a Mint/MATE live CD/USB may wish to exit *Nano* and use MATE's text editor *Pluma* which is a clone of the popular *gEdit*; however, *Nano* is certainly up to the task if you don't mind becoming familiar with its UI.

If you're running from the live CD, the *changelog* file that's created will look something like this:

```
pipewrench (1.0-1) UNRELEASED; urgency=low

  * Initial release. (Closes: #XXXXXX)

 -- Live session user <mint@mint>  Tue, 13 Aug 2013 19:14:39 +0000
```

The “-1” that we added to the software's version in running the dch command can be seen on the first line of the generated *changelog* file. It represents the Debian package version. So according to the *changelog*, the package we're creating will be of packaging version 1. If we were to later make changes to the packaging (i.e., a change made inside of the debian subdirectory), we'd have to “bump” this latter part of the overall version number to “-2”.

For the purposes of this tutorial, leave “UNRELEASED” as is. This field would normally indicate to the packaging tool where the newly created Debian binary package should be uploaded to. By leaving this as “UNRELEASED”, no upload can be performed and this is what we want here.

The “urgency=low” is fine for our purposes. Leave it as is.

The “(Closes: #XXXXXX)” field is normally a means of automating the closing of the “bug” that an updated Debian package is meant to fix (if indeed the package was intended as a patch). For our purposes, this text can either be removed or it can be ignored. Your choice. If ignored, it will simply result in a warning later that won't stop us from being able to install the package.

As for the last line, the packaging tool attempts to guess who the packager is and in my case it thinks that I'm “Live session user” and that my email address is “mint@mint”. Replacing the former with your full name and the latter with your email will eliminate another warning that would otherwise be generated later.

So, that's it for the *changelog* file. You can save the changes to it and close it.

Packaging File: compat

The *compat* file indicates what version of debhelper is required in order to build a package. According to the tutorial that this one is based on, the *compat* file should contain just the digit “8” in it so, from the terminal, run the following commands:

```
cd ~/Desktop/PipeWrench-Packaging/pipewrench-1.0/debian
echo 8 >compat
```

This will create the *compat* file. Notice that there's a space following the digit, “8”. Without it, our *compat* file would still be created, but it would have no contents.

Packaging File: control

The control file provides some details concerning the source and binary packages. As such, it is split into two main sections: the first one involving the source package and the latter one, the binary package.

Following is the control file that we will use:

```
Source: pipewrench
Section: devel
Priority: optional
Maintainer: Barry Block <bwb@fireflysoftware.com>
Build-Depends: debhelper (>= 7.0.50~),
               cli-common-dev (>= 0.7),
               libmono-cil-dev,
               mono-devel (>= 2.6.7),
               mono-xbuild,
               libgtk2.0-cil-dev (>= 2.12.10),
               libglade2.0-cil-dev (>= 2.12.10)
Standards-Version: 3.9.5

Package: pipewrench
Architecture: any
Depends: ${cli:Depends}, ${misc:Depends}
Description: Easily transform text into other forms using stackable filters
 PipeWrench is a high-level text scripting tool that allows you to easily
 transform text, (i.e. lists, command output, HTML, config files, log files,
 source code, CSV data, etc.) from one form to another with relative ease.
 In a PipeWrench script, there's no need to create conditional or looping
 constructs or even to declare variables. You simply "stack" filters to get
 the results you want. And when you're done building your "pipe script"
 using PipeWrench's editor, you can export it for use at the command line
 (or in shell scripts).
```

The *Source* field (required) specifies the name of the source package.

The *Section* field specifies the section of the distribution that the package goes into. Here, “devel” translates to “programmer tools”.

The *Priority* field is “the priority of the package (one of 'required', 'important', 'standard', 'optional' or 'extra'). In general a package is 'optional' unless it's 'essential' for a standard functioning system, i.e., booting or networking functionality. A package should be 'extra' rather than 'optional' if it conflicts with another 'optional' package, or if it's not intended to be used on a standard desktop installation. Notable example of 'extra' packages are debugging packages.”[7]

The *Maintainer* field (required) specifies who is responsible for maintaining the package.

The *Build-Depends* field specifies the packages that must be installed in order to build this package. Note here that the lines are indented using *spaces*.

The *Standards-Version* field specifies the most recent version of the Debian policy manual for which the package complies.

The *Package* field (required) specifies the name of the binary package.

7 Introduction to Debian Packaging; https://wiki.debian.org/IntroDebianPackaging#Introduction_to_Debian_Packaging

The *Architecture* field (required) specifies which hardware architectures the binary package is expected to work on. Specific architectures like i386 or amd64 can be listed; however, it is common to use two special designations:

- A value of "any" means that the software is coded in a portable fashion so that it will function on all architectures but a separate binary package must still be built for each architecture.
- A value of "all" means that the software will run on any architecture but only one package needs to be built that can be used across all of them.

The *Depends* field specifies the packages that must be installed in order to run the software that this package installs. You can manually determine and list each required package individually but that can be tedious and error-prone. For an application running on Mono, you can simply specify "${cli:Depends}, ${misc:Depends}" as the value of the *Depends* field and most, if not all of the application's dependencies will be automatically determined for you.

The *Description* field (required) provides both short and long descriptions of the package, meant to be of use to end-users of the package. This description is what's displayed when you install the binary package using gdebi or similar GUI. Be mindful of the formatting of the *control* file's description field. On the same line as "Description:" is the short description. All lines following it are part of the long description and each of those are indented by a single *space*. Note also that sentences are themselves followed by a single space and NOT by two spaces.

For additional information regarding the *control* file's fields, please see the Debian policy manual at http://www.debian.org/doc/debian-policy.

Now, back to the building of our package. You need not enter all of the *control* file's fields by hand. Simply copy the above text into your own *control* file created beneath the *debian* directory.

Note: Be sure to format the copied text *exactly* the same as you see above. Text copied from PDF files typically excludes blank lines and any leading white space.[8]

Packaging File: copyright

The *copyright* file is an important part of a Debian package but as this tutorial is concerned only with the technical aspects of building a Debian package, we'll simply create an empty *copyright* file. From the terminal, run the following commands:

```
cd ~/Desktop/PipeWrench-Packaging/pipewrench-1.0/debian
touch copyright
```

This creates an empty file named "copyright".

8 I don't know if this is an inherent problem of the PDF structure itself, or an issue involving the particular PDF viewer that's used; all I know is that it's annoying.

Packaging File: install

The *install* file is used to instruct the build process where to "install" each file of the package's payload. Each line of the file contains information for copying a file from somewhere within the source subtree to a destination beneath /usr.

For PipeWrench, the *install* file contains the following lines:

```
Extras/Linux/pipewrench usr/bin
Extras/Linux/pcl usr/bin
Extras/Linux/pipewrench.desktop usr/share/applications
Extras/pipewrench.png usr/share/icons
LinGUI/bin/Release/PipeWrench.exe usr/lib/pipewrench
PCL/bin/Release/PCL.exe usr/lib/pipewrench
Shared/bin/Release/Shared.dll usr/lib/pipewrench
README usr/share/doc/pipewrench
Extras/userguide.pdf usr/share/pipewrench
Extras/Demos usr/share/pipewrench
```

As you can see, the file is processed relative to the main (top-level) source directory (which in our case is the one containing the PipeWrench.sln file).

Again, there's no need to type the above lines into your *install* file. Simply copy the above text into it.

Packaging File: <package>.links

The *<package>.links* file—which is used by debhelper's *dh_link*—is used to create *symlinks* during package installation. The one for PipeWrench should contain the following lines:

```
usr/share/pipewrench/userguide.pdf usr/lib/pipewrench/userguide.pdf
usr/share/pipewrench/Demos usr/lib/pipewrench/Demos
```

Each line represents a *symlink* that needs to be created. The first path on each line is the file (or directory) being linked to and the second path is that of the *symlink* that's being created.

Simply create a *pipewrench.links* file beneath the *debian* directory and copy the above text into it.

Packaging File: <package>.manpages

The *<package>.manpages* file—which is used by debhelper's *dh_installman*—is used to install a package's manual pages during package installation. The one for PipeWrench should contain the following lines:

```
Extras/Linux/pipewrench.1
Extras/Linux/pcl.1
```

As you can see, each line is just a path to a "man page" file and like with the *install* file, the paths are specified relative to the main (top-level) source directory.

Simply create a *pipewrench.manpages* file beneath the *debian* directory and copy the above text into it.

Packaging File: rules

The *rules* file is a make file that directs the building of your package. Following is the *rules* file I'm using to package PipeWrench:

```
#!/usr/bin/make -f

%:
	dh $@ --with=cli

override_dh_auto_build:
	xbuild $(CURDIR)/PipeWrench.sln /p:Configuration=Release

override_dh_auto_clean:
	xbuild $(CURDIR)/PipeWrench.sln /p:Configuration=Release /t:Clean
```

As the *rules* file is a make file, each line ending with a colon (:) is called a “target” and all of the indented lines that follow it represent the “recipe” for creating that target and each of them must be indented by just a single *tab* character. Again, you can just copy the above text into your own *rules* file. Of course, when packaging your own software, you'll want to edit “PipeWrench.sln” to whatever is appropriate in your case.

Note: Like with the *control* file, the *rules* file must be formatted carefully. Again, when the above text is copied from this PDF, *any blank lines and leading whitespace might get excluded and need to be restored in the resulting text*. The blank lines aren't absolutely necessary in a make file like this; however, the indented lines following each *target* must be indented using just a single *tab* character.

Packaging File: source/format

The last file we need to create beneath the *debian* directory is the *format* file contained within the *source* subdirectory. It merely specifies the version of the source package's format. All you need to do here is run the following commands from the terminal:

```
cd ~/Desktop/PipeWrench-Packaging/pipewrench-1.0/debian
mkdir source
echo "3.0 (quilt)" >source/format
```

Building the Package

Hopefully, at this juncture we have a properly-constructed *debian* subdirectory to support building of our package. An important next step that we want to take here is to create a back up of our work done thus far. This is made important by the fact that any build attempt can conceivably “dirty” our working subtree, requiring us to perform a clean up before we can attempt another build and we don't need the extra hassle when we're working out problems that might arise during the process of building a package. So, let's make an archive of all of our work done by running the following commands from the terminal:

```
cd ~/Desktop
tar -cvzf PipeWrench-Packaging.tar.gz PipeWrench-Packaging
```

So now we have something to revert back to if we encounter any troubles. I've done my best to limit how often you will need to revert your working subtree back to the original state; however, in a normal situation, mistakes made in setting up the *debian* directory (or even those directories source directories outside of it) can cause you to need to start over. Here's the thing: *When you do encounter a mistake made, don't make the changes directly to your working subtree—make them to your back up of it.* That way, you're not having to re-do prior changes made each time you encounter a problem that needs to be fixed.

It's now time to try our first build! The build needs to be initiated from the main (top-level) source directory which again is the one containing *PipeWrench.sln* in it. From the terminal, run the following commands:

```
cd ~/Desktop/PipeWrench-Packaging/pipewrench-1.0
debuild -us -uc
```

Unless your system was already set up to build a GTK sharp application, chances are that you'll see a message here complaining about missing build dependencies. The message should include a line similar to this one:

```
dpkg-checkbuilddeps: Unmet build dependencies: cli-common-dev (>= 0.7)
libmono-cil-dev mono-devel (>= 2.6.7) mono-xbuild libgtk2.0-cil-dev (>=
2.12.10) libglade2.0-cil-dev (>= 2.12.10)
```

Recall, in the *control* file, we specified a number of packages that have to be installed on our system in order for us to build our package. These are the packages listed in the *build-depends* field. We just simply need to install them. From the terminal, you'll need to run a command of this general form:

```
sudo apt-get install <list of missing packages delimited by spaces>
```

Of course, "<list of missing packages delimited by spaces>" needs to be replaced by the actual list of packages that were reported as not being installed on *your* system. In practice, you only need to copy the list of reported missing dependencies and edit out everything in parentheses.

So, for example, if the above error message applies in your case (and it may, if you're running from the Linux Mint/MATE live CD), you'd want to run the following two commands (individually) from the terminal:

```
sudo apt-get install cli-common-dev libmono-cil-dev mono-devel
```

And now, the second one:

```
sudo apt-get install mono-xbuild libgtk2.0-cil-dev libglade2.0-cil-dev
```

Note: Of course, these two commands could've been combined into a single one here but the resulting command would have "wrapped" onto a second line and you wouldn't have been able to copy it into the terminal without problems (the wrapped part of the command would be interpreted by the shell as an entirely distinct command, which wouldn't be too cool. And, as luck would have it, you can't even copy the two commands together into the terminal to run them—which is why I separated them here—because attempting that results in neither command running at all).

Now, after having encountered this particular problem, (that being missing dependencies) it's probably not necessary to revert back to the original working subtree; but, for practice sake, why not do it anyway? Go ahead and rename the *PipeWrench-Packaging* directory so that it's out of our way by running the following commands from the terminal:

```
cd ~/Desktop
mv PipeWrench-Packaging PipeWrench-Packaging-old
```

Now extract the archive, PipeWrench-*Packaging.tar.gz* that should still be residing in the Desktop directory where we created it. From the terminal, run this command:

```
tar -xvzf PipeWrench-Packaging.tar.gz
```

Now the working subtree is back to a pristine starting point and we can attempt another build:

```
cd ~/Desktop/PipeWrench-Packaging/pipewrench-1.0
debuild -us -uc
cd ..
```

This time hopefully, all you will see generated are warning messages and not errors. If, at the end of the output generated, *lintian* didn't generate any actual errors (all of its messages are preceded by "W:" and not "E:"), then there ought to be a freshly-built debian package waiting for you in the current directory. You can quickly check this from the terminal by running the following command:

```
ls -l *.deb
```

If you see a file listed named *pipewrench_1.0-1_xxx.deb*, *(where "xxx" is representative of the architecture of the Linux live CD that you're running from—either "i386" or "amd64")*, then congratulations, *you've built a Debian package of a C#/Mono application starting from source!*

Installing and Running the Package

Ok, perhaps it's a little early to celebrate... Let's see if the package will install and run. To install it, run the following command from the terminal (after first changing "xxx" to either "i386" or "amd64" as per your setup):

```
sudo gdebi pipewrench_1.0-1_xxx.deb
```

If the install goes according to plan then you ought to find PipeWrench in the desktop environment's main menu. If you're running Linux Mint/MATE edition, you can just press the "Windows" key and type "pipe" and it should appear on the right side of the menu. Just down arrow to it and press Enter to run it.

Likewise, the command line component of PipeWrench, (PCL) should be accessible from the terminal. To run it, enter the following into the terminal:

```
pcl
```

You should be presented with PCL's help screen.

Well, that about wraps it up for packaging a Mono/C# application on a Debian-based system. Though this tutorial focuses on a somewhat narrow example of packaging and it certainly leaves out many of the finer points of Debian packaging, hopefully there's enough here that it will help put you on the right track to packaging your own C# applications!

If you found this tutorial to be helpful, please leave a comment on Linux Mint's community portal at http://community.linuxmint.com/tutorial/view/1373.