PKGBUILD Style Guide

August 13, 2014

Contents

Introduction

> "Using a simple tool to solve a complex problem does not result in a simple solution." – Larry Wall

The PKGBUILD is a file containing a set of instructions sourced by makepkg to build software packages for the pacman package manager.

`makepkg` is the main implementation used to generate packages, it is written in the bash shell scripting and so too are PKGBUILDs. This allows for a high level of expression and flexibility while remaining relatively easy to implement.

However, due to this expression many PKGBUILDs are sometimes abused with arcane or confusing bash scripting constructs often written by people who aren't familiar with the bash shell scripting language.

This guide is my attempt to impose what I personally believe to be a cleaner template for PKGBUILDs along with a discussion for each topic in an effort to make PKGBUILDs better and more succinct.

Note: While the PKGBUILD is considered merely a means to an end, which is certainly fair, I believe that a clearer approach will make them both easier to maintain and change if circumstances change without being overly complex.

Warning: This is not complete and subject to change, if you disagree with a certain design choice I have made, please feel free to present any arguments to sway me to your point of view.

The Ideal PKGBUILD

> "A map of the world that does not include Utopia is not worth even glancing at." – Oscar Wilde

Below is a reasonably complete PKGBUILD which I consider embodies the ideals I want to impose. Although not necessary the spacing of common purpose variables makes reading dense PKGBUILDs easier.

```
pkgname=antimicro-git
pkgver=2.5.r11.g09f8ce7
pkgrel=1

pkgdesc='Map keyboard and mouse actions to gamepad buttons, inspired by qjoypad.'
url='https://github.com/Ryochan7/antimicro'
arch=('i686' 'x86_64')
license=('GPL3')

depends=('libxtst' 'qt5-base' 'sdl2' 'libxkbcommon-x11')
makedepends=('git' 'cmake' 'qt5-tools')

provides=('antimicro')
conflicts=('antimicro')

source=('git://github.com/Ryochan7/antimicro')
install='antimicro-git.install'

md5sums=('SKIP')

pkgver() {
    cd antimicro
    git describe | sed 's/-/.r/; s/-/./'
}

build() {
    cd antimicro
    cmake -DCMAKE_INSTALL_PREFIX=/usr -DUSE_SDL_2=ON
    make
}

package() {
    cd antimicro
    make DESTDIR="$pkgdir" install
}
```

Dealing with `git` submodules

> Programs must be written for people to read, and only incidentally for machines to execute. – Hal Abelson

Often when building packages from git sources you can encounter submodules. These can be problematic if not handled correctly particularly if they are large repositories as each time you run `makepkg` with a naïeve PKGBUILD it will re-initialise and download each one, each time.

A good strategy to deal with this is including each submodule as a separate entry in `source=()` and then update the main repositories submodule configuration to point at the local clones in the `prepare()` function.

Case study: PPSSPP

PPSSPP is a HLE (High Level Emulator) emulating the PSP. It contains a number of submodules and a custom fork of ffmpeg which is over 340M in size!

A naïeve solution may look something like this:

```
source=('git://github.com/hrydgard/ppsspp.git')

prepare() {
    cd ppsspp
    git submodule update --init --recursive
    # Potentially more operations such as checking out branches and pulling
}
```

It's pretty unrealistic to clone over 350M of data for every invocation of `makepkg` so lets fix this.

We begin by looking at PPSSPP's `.gitmodules` file and see the following:

```
[submodule "native"]
    path = native
    url = https://github.com/hrydgard/native.git
[submodule "pspautotests"]
    path = pspautotests
    url = https://github.com/hrydgard/pspautotests.git
[submodule "lang"]
    path = lang
    url = https://github.com/hrydgard/ppsspp-lang.git
[submodule "ffmpeg"]
    path = ffmpeg
```

```
    url = https://github.com/hrydgard/ppsspp-ffmpeg.git
[submodule "dx9sdk"]
    path = dx9sdk
    url = https://github.com/hrydgard/minidx9.git
[submodule "redist"]
    path = redist
    url = https://github.com/hrydgard/ppsspp-redist.git
```

We can immediately see that not only are there some fairly large repositories but that we don't even need half of these on Linux!

After some investigation we might find that the only needed submodules for us is `native`, `lang` and `ffmpeg` as we don't need Visual C 2010 Redistribution files for an installer (`redist`), DirectX9 SDK for their buildbot (`dx9sdk`) or the development tools like `pspautotests`.

So lets flesh out our naïeve solution and add those submodules to the `source=()` array.

```
source=('git://github.com/hrydgard/ppsspp.git'
        'git://github.com/hrydgard/native.git'
        'git://github.com/hrydgard/ppsspp-lang.git'
        'git://github.com/hrydgard/ppsspp-ffmpeg.git')

prepare() {
    cd ppsspp
    git submodule update --init --recursive
    # Potentially more operations such as checking out branches and pulling
}
```

Oops, we've now managed to clone everything twice and more! This is because we haven't fixed the `prepare()` function to switch the location for our newly cloned submodules.

Git provides a git-config(1) command which lets us intelligently reconfigure the repositories submodules.

With this in mind we can manually adjust each of the required submodules to use `"$srcdir"/<submodule>` and then update them.

```
source=('git://github.com/hrydgard/ppsspp.git'
        'git://github.com/hrydgard/native.git'
        'git://github.com/hrydgard/ppsspp-lang.git'
        'git://github.com/hrydgard/ppsspp-ffmpeg.git')

prepare() {
    cd ppsspp
    # First initialise the submodules.
```

```
    git submodule init

    # Each of the clones are included under "$srcdir" (as everything in the
    # `source=()` is) so we simply need to adjust the submodules to use them
    # instead.
    git config submodule.native.url "$srcdir"/native
    git config submodule.lang.url "$srcdir"/ppsspp-lang
    git config submodule.ffmpeg.url "$srcdir"/ppsspp-ffmpeg

    # Now that we've configured the needful submodules let's update them.
    # Note: You need to be explicit here otherwise it will attempt to update
    #       the other unecessary submodules as well!
    git submodule update native lang ffmpeg
}
```

Some git repositories can be a little tricky with regards to where exactly the submodules are found.

For example blender keeps all of their submodules under `release/*/submodule` which would thus require using the following `git-config` to correctly set the new location:

```
git submodule init
git config submodule.release/scripts/addons.url "$srcdir"/addons
git submodule update release/scripts/addons
```

And that's all there is to it. Submodules dealt with.

The Do's and Don'ts

The pkgver

Don't include any hyphens.

Hyphens are used by pacman as markers to cut the package name up into logical blocks. For example consider the following with a pkgname of foo-bar-baz:

```
foo-bar-baz-pkgver-pkgrel-carch.pkg.tar.xz
```

1) Read backwards from the end until the first . then until the first -. This becomes the carch.
2) Then read until the next -. This becomes the pkgrel.
3) finally read until the next - again. This is the pkver

Due to this pkgnames can contain hyphens but versions cannot.

The code responsible for this can be found at lib/libalpm/util.c:1069

The pkgver and VCS

Software from version control such as git, mercurial (hg) and others no one cares about anymore can be tricky to deal with as they never contain a constant version.

Typically using any good version control software you can easily get a nicely formatted result, e.g. with git describe:

```
$ git describe
v0.5.3-54-g8274f4
```

What you're looking at follows a simple formula: the v0.5.3 is the latest tag followed by 54 which indicates the number of revisions since that tag. Lastly is the latest commit hash, iconically prefixed with a g as git does.

Thus what we get is essentially: (latest-tag)-(revisions-since-tag)-(latest-commit-hash)

To deal with pacman not allowing hyphens we can clean up this information to be even more unambiguous by replacing the first hyphen with .r and the second with . while stripping the leading v; thus we want something resembling 0.5.3.r54.g8274f4. To achieve this a simple sed command can be used:

```
$ get describe | sed 's/^v//; s/-/.r/; s/-/./'
0.5.3.r54.g8274f4
```

If the software does not have a tag then use the form `(total-revisions).(latest-commit-hash)`.

```
$ printf 'r%s.%s' "$(git rev-list --count HEAD)" "$(git describe --always)"
r952.g8274f4
```

The benefit of this is that when upstream decides to add a release tag the new version will be recognised as newer by `pacman`. You can test this using the `vercmp` tool which prints `1` if the left-hand-side (LHS) is greater than the right (RHS) or `0` if they are equal.:

```
$ vercmp 0.5.3.r54.g8274f4 r952.g8274f4
1
```

Let's compare this to using a different form. Instead of using the semantically meaningful `.r` notation we will separate the segments using `+` instead.

```
$ vercmp 0.5.3+54+g8274f4 952+g8274f4
-1
```

Oh dear, this means we'll now have to rely on a *permanent* addition of an epoch or feel the wrath of users complaining about never getting any newer updates even though upstream is well ahead of your lousy old package!

Other Version Control Software

Unfortunately not everything uses git and we need to deal with them as well.

Let's use the vim for our example.

```
$ hg log -r "." --template \
    "{sub('-', '.', strip(latesttag, 'v'))}.r{latesttagdistance}.{node|short}"
7.4.397.r1.2a798dca16bf
```

Holy sundials batman what is this mess?! Don't ask me... Oh, I'm supposed to be explaining this...

What's going on here is firstly `hg log -r "."` which tells hg to only look at the latest revision, then we use a custom template which first substitutes our hyphens for periods, just as before, it then strips off the leading `v`.

Next we use `{latesttagdistance}` to fetch the number of revisions since the latest tag followed by the `{node}` which has been modified with `short` to get the latest commit.

Not convinced? Here's another example from using pentadactyl.

```
$ hg log -r "." --template \
    "{sub('-', '.', strip(latesttag, 'pentadactyl-'))}.r{latesttagdistance}.{node|short}"
1.1.r36.53053d10138a
```

This time we needed to strip off a leading `pentadactyl-` instead of a simple `v`.

`bzr`, `svn`, `cvs`, `darcs`, `rcs` and `sccs` can get fucked.