RPM HOWTO (RPM at Idle)

Donnie Barnes, djb@redhat.com V2.0, April 8, 1997

Contents

1 Introduction

RPM is the **R**ed Hat **P**ackage **M**anager. While it does contain Red Hat in the name, it is completely intended to be an open packaging system available for anyone to use. It allows users to take source code for new software and package it into source and binary form such that binaries can be easily installed and tracked and source can be rebuilt easily. It also maintains a database of all packages and their files that can be used for verifying packages and querying for information about files and/or packages.

Red Hat Software encourages other distribution vendors to take the time to look at RPM and use it for their own distributions. RPM is quite flexible and easy to use, though it provides the base for a very extensive system. It is also completely open and available, though we would appreciate bug reports and fixes. Permission is granted to use and distribute RPM royalty free under the GPL.

More complete documentation is available on RPM in the book by Ed Bailey, *Maximum RPM*. That book is available for download or purchase at *www.redhat.com* (`http://www.redhat.com`).

2 Overview

First, let me state some of the philosophy behind RPM. One design goal was to allow the use of "pristine" sources. With RPP (our former packaging system of which *none* of RPM is derived), our source packages were the "hacked" sources that we built from. Theoretically, one could install a source RPP and then `make` it with no problems. But the sources were not the original ones, and there was no reference as to what changes we had to make to get it to build. One had to download the pristine sources separately. With RPM, you have the pristine sources along with a patch that we used to compile from. We see this as a big advantage. Why? Several reasons. For one, if a new version of a program comes out, you don't necessarily have to start from scratch to get it to compile under RHL. You can look at the patch to see what you *might* need to do. All the compile-in defaults are easily visible this way.

RPM is also designed to have powerful querying options. You can do searches through your entire database for packages or just certain files. You can also easily find out what package a file belongs to and where it came from. The RPM files themselves are compressed archives, but you can query individual packages easily and *quickly* because of a custom binary header added to the package with everything you could possibly need to know contained in uncompressed form. This allows for *fast* querying.

Another powerful feature is the ability to verify packages. If you are worried that you deleted an important file for some package, just verify it. You will be notified of any anomalies. At that point, you can reinstall the package if necessary. Any config files that you had are preserved as well.

We would like to thank the folks from the BOGUS distribution for many of their ideas and concepts that are included in RPM. While RPM was completely written by Red Hat Software, its operation is based on code written by BOGUS (PM and PMS).

3 General Information

3.1 Acquiring RPM

The best way to get RPM is to install Red Hat Linux. If you don't want to do that, you can still get and use RPM. It can be acquired from *ftp.redhat.com* (`ftp://ftp.redhat.com/pub/redhat/code/rpm`).

3.2 RPM Requirements

The main requirement to run RPM is cpio 2.4.2 or greater. While this system is intended for use with Linux, it may very well be portable to other Unix systems. It has, in fact, been compiled on SunOS, Solaris, AIX, Irix, AmigaOS, and others. Be warned, the binary packages generated on a different type of Unix system will not be compatible.

Those are the minimal requirements to install RPMs. To build RPMs from source, you also need everything normally required to build a package, like `gcc`, `make`, etc.

4 Using RPM

In its simplest form, RPM can be used to install packages:

```
rpm -i foobar-1.0-1.i386.rpm
```

The next simplest command is to uninstall a package:

```
rpm -e foobar
```

One of the more complex but *highly* useful commands allows you to install packages via FTP. If you are connected to the net and want to install a new package, all you need to do is specify the file with a valid URL, like so:

```
rpm -i ftp://ftp.pht.com/pub/linux/redhat/rh-2.0-beta/RPMS/foobar-1.0-1.i386.rpm
```

Please note, that RPM will now query and/or install via FTP.

While these are simple commands, rpm can be used in a multitude of ways as seen from the `Usage` message:

```
RPM version 2.3.9
Copyright (C) 1997 - Red Hat Software
This may be freely redistributed under the terms of the GNU Public License

usage: rpm {--help}
       rpm {--version}
       rpm {--initdb}   [--dbpath <dir>]
       rpm {--install -i} [-v] [--hash -h] [--percent] [--force] [--test]
                        [--replacepkgs] [--replacefiles] [--root <dir>]
                        [--excludedocs] [--includedocs] [--noscripts]
                        [--rcfile <file>] [--ignorearch] [--dbpath <dir>]
                        [--prefix <dir>] [--ignoreos] [--nodeps]
                        [--ftpproxy <host>] [--ftpport <port>]
                        file1.rpm ... fileN.rpm
       rpm {--upgrade -U} [-v] [--hash -h] [--percent] [--force] [--test]
                        [--oldpackage] [--root <dir>] [--noscripts]
                        [--excludedocs] [--includedocs] [--rcfile <file>]
                        [--ignorearch]  [--dbpath <dir>] [--prefix <dir>]
                        [--ftpproxy <host>] [--ftpport <port>]
                        [--ignoreos] [--nodeps] file1.rpm ... fileN.rpm
       rpm {--query -q} [-afpg] [-i] [-l] [-s] [-d] [-c] [-v] [-R]
                        [--scripts] [--root <dir>] [--rcfile <file>]
                        [--whatprovides] [--whatrequires] [--requires]
                        [--ftpuseport] [--ftpproxy <host>] [--ftpport <port>]
                        [--provides] [--dump] [--dbpath <dir>] [targets]
       rpm {--verify -V -y} [-afpg] [--root <dir>] [--rcfile <file>]
                        [--dbpath <dir>] [--nodeps] [--nofiles] [--noscripts]
                        [--nomd5] [targets]
       rpm {--setperms} [-afpg] [target]
       rpm {--setugids} [-afpg] [target]
       rpm {--erase -e} [--root <dir>] [--noscripts] [--rcfile <file>]
                        [--dbpath <dir>] [--nodeps] [--allmatches]
                        package1 ... packageN
       rpm {-b|t}[plciba] [-v] [--short-circuit] [--clean] [--rcfile  <file>]
                        [--sign] [--test] [--timecheck <s>] specfile
       rpm {--rebuild} [--rcfile <file>] [-v] source1.rpm ... sourceN.rpm
       rpm {--recompile} [--rcfile <file>] [-v] source1.rpm ... sourceN.rpm
       rpm {--resign} [--rcfile <file>] package1 package2 ... packageN
       rpm {--addsign} [--rcfile <file>] package1 package2 ... packageN
       rpm {--checksig -K} [--nopgp] [--nomd5] [--rcfile <file>]
                           package1 ... packageN
       rpm {--rebuilddb} [--rcfile <file>] [--dbpath <dir>]
       rpm {--querytags}
```

You can find more details on what those options do in the RPM man page.

5 Now what can I *really* do with RPM?

RPM is a very useful tool and, as you can see, has several options. The best way to make sense of them is to look at some examples. I covered simple install/uninstall above, so here are some more examples:

- Let's say you delete some files by accident, but you aren't sure what you deleted. If you want to verify your entire system and see what might be missing, you would do:

  ```
  rpm -Va
  ```

- Let's say you run across a file that you don't recognize. To find out which package owns it, you would do:

  ```
  rpm -qf /usr/X11R6/bin/xjewel
  ```

  The output would be:

  ```
  xjewel-1.6-1
  ```

- You find a new koules RPM, but you don't know what it is. To find out some information on it, do:

  ```
  rpm -qpi koules-1.2-2.i386.rpm
  ```

  The output would be:

  ```
  Name        : koules                      Distribution: Red Hat Linux Colgate
  Version     : 1.2                               Vendor: Red Hat Software
  Release     : 2                             Build Date: Mon Sep 02 11:59:12 1996
  Install date: (none)                        Build Host: porky.redhat.com
  Group       : Games                         Source RPM: koules-1.2-2.src.rpm
  Size        : 614939
  Summary     : SVGAlib action game with multiplayer, network, and sound support
  Description :
  This arcade-style game is novel in conception and excellent in execution.
  No shooting, no blood, no guts, no gore.  The play is simple, but you
  still must develop skill to play.  This version uses SVGAlib to
  run on a graphics console.
  ```

- Now you want to see what files the koules RPM installs. You would do:

  ```
  rpm -qpl koules-1.2-2.i386.rpm
  ```

  The output is:

  ```
  /usr/doc/koules
  /usr/doc/koules/ANNOUNCE
  /usr/doc/koules/BUGS
  /usr/doc/koules/COMPILE.OS2
  /usr/doc/koules/COPYING
  ```

```
/usr/doc/koules/Card
/usr/doc/koules/ChangeLog
/usr/doc/koules/INSTALLATION
/usr/doc/koules/Icon.xpm
/usr/doc/koules/Icon2.xpm
/usr/doc/koules/Koules.FAQ
/usr/doc/koules/Koules.xpm
/usr/doc/koules/README
/usr/doc/koules/TODO
/usr/games/koules
/usr/games/koules.svga
/usr/games/koules.tcl
/usr/man/man6/koules.svga.6
```

These are just several examples. More creative ones can be thought of really easy once you are familiar with RPM.

6 Building RPMs

Building RPMs is fairly easy to do, especially if you can get the software you are trying to package to build on its own.

The basic procedure to build an RPM is as follows:

- Make sure your `/etc/rpmrc` is setup for your system.
- Get the source code you are building the RPM for to build on your system.
- Make a patch of any changes you had to make to the sources to get them to build properly.
- Make a spec file for the package.
- Make sure everything is in its proper place.
- Build the package using RPM.

Under normal operation, RPM builds both binary and source packages.

6.1 The rpmrc File

Right now, the only configuration of RPM is available via the `/etc/rpmrc` file. An example one looks like:

```
require_vendor: 1
distribution: I roll my own!
require_distribution: 1
topdir: /usr/src/me
vendor: Mickiesoft
```

```
packager:  Mickeysoft Packaging Account <packages@mickiesoft.com>

optflags: i386 -O2 -m486 -fno-strength-reduce
optflags: alpha -O2
optflags: sparc -O2

signature: pgp
pgp_name: Mickeysoft Packaging Account
pgp_path: /home/packages/.pgp

tmppath: /usr/tmp
```

The `require_vendor` line causes RPM to require that it find a vendor line. This can come from the `/etc/rpmrc` or from the header of the spec file itself. To turn this off, change the number to `0`. The same holds true for the `require_distribution` and `require_group` lines.

The next line is the `distribution` line. You can define that here or later in the header of the spec file. When building for a particular distribution, it's a good idea to make sure this line is correct, even though it is not required. The `vendor` line works much the same way, but can be anything (ie. Joe's Software and Rock Music Emporium).

RPM also now has support for building packages on multiple architectures. The `rpmrc` file can hold an "optflags" variable for building things that require architecture specific flags when building. See later sections for how to use this variable.

In addition to the above macros, there are several more. You can use:

```
rpm --showrc
```

to find out how your tags are set and what all the available flags are.

6.2 The Spec File

We'll begin with discussion of the spec file. Spec files are required to build a package. The spec file is a description of the software along with instructions on how to build it and a file list for all the binaries that get installed.

You'll want to name your spec file according to a standard convention. It should be the package name-dash-version number-dash-release number-dot-spec.

Here is a small spec file (vim-3.0-1.spec):

```
Summary: ejects ejectable media and controls auto ejection
Name: eject
Version: 1.4
Release: 3
Copyright: GPL
Group: Utilities/System
Source: sunsite.unc.edu:/pub/Linux/utils/disk-management/eject-1.4.tar.gz
```

```
Patch: eject-1.4-make.patch
Patch1: eject-1.4-jaz.patch
%description
This program allows the user to eject media that is autoejecting like
CD-ROMs, Jaz and Zip drives, and floppy drives on SPARC machines.

%prep
%setup
%patch -p1
%patch1 -p1

%build
make RPM_OPT_FLAGS="$RPM_OPT_FLAGS"

%install
install -s -m 755 -o 0 -g 0 eject /usr/bin/eject
install -m 644 -o 0 -g 0 eject.1 /usr/man/man1

%files
%doc README COPYING ChangeLog

/usr/bin/eject
/usr/man/man1/eject.1
```

6.3 The Header

The header has some standard fields in it that you need to fill in. There are a few caveats as well. The fields must be filled in as follows:

- `Summary:` This is a one line description of the package.
- `Name:` This must be the name string from the rpm filename you plan to use.
- `Version:` This must be the version string from the rpm filename you plan to use.
- `Release:` This is the release number for a package of the same version (ie. if we make a package and find it to be slightly broken and need to make it again, the next package would be release number 2).
- `Icon:` This is the name of the icon file for use by other high level installation tools (like Red Hat's "glint"). It must be a gif and resides in the SOURCES directory.
- `Source:` This line points at the HOME location of the pristine source file. It is used if you ever want to get the source again or check for newer versions. Caveat: The filename in this line MUST match the filename you have on your own system (ie. don't download the source file and change its name). You can also specify more than one source file using lines like:

  ```
  Source0: blah-0.tar.gz
  Source1: blah-1.tar.gz
  Source2: fooblah.tar.gz
  ```

These files would go in the SOURCES directory. (The directory structure is discussed in a later section, "The Source Directory Tree".)

- Patch: This is the place you can find the patch if you need to download it again. Caveat: The filename here must match the one you use when you make YOUR patch. You may also want to note that you can have multiple patch files much as you can have multiple sources.] You would have something like:

```
Patch0: blah-0.patch
Patch1: blah-1.patch
Patch2: fooblah.patch
```

  These files would go in the SOURCES directory.

- Copyright: This line tells how a package is copyrighted. You should use something like GPL, BSD, MIT, public domain, distributable, or commercial.

- BuildRoot: This line allows you to specify a directory as the "root" for building and installing the new package. You can use this to help test your package before having it installed on your machine.

- Group: This line is used to tell high level installation programs (such as Red Hat's "glint") where to place this particular program in its hierarchical structure. The group tree currently looks something like this:

```
Applications
    Communications
    Editors
        Emacs
    Engineering
    Spreadsheets
    Databases
    Graphics
    Networking
    Mail
    Math
    News
    Publishing
        TeX
Base
    Kernel
Utilities
    Archiving
    Console
    File
    System
    Terminal
    Text
Daemons
Documentation
X11
    XFree86
```

```
        Servers
    Applications
        Graphics
        Networking
    Games
        Strategy
        Video
    Amusements
    Utilities
    Libraries
    Window Managers
Libraries
Networking
    Admin
    Daemons
    News
    Utilities
Development
    Debuggers
    Libraries
        Libc
    Languages
        Fortran
        Tcl
    Building
    Version Control
    Tools
Shells
Games
```

- `%description` It's not really a header item, but should be described with the rest of the header. You need one description tag per package and/or subpackage. This is a multi-line field that should be used to give a comprehensive description of the package.

6.4 Prep

This is the second section in the spec file. It is used to get the sources ready to build. Here you need to do anything necessary to get the sources patched and setup like they need to be setup to do a `make`.

One thing to note: Each of these sections is really just a place to execute shell scripts. You could simply make an `sh` script and put it after the `%prep` tag to unpack and patch your sources. We have made macros to aid in this, however.

The first of these macros is the `%setup` macro. In its simplest form (no command line options), it simply unpacks the sources and `cd`'s into the source directory. It also takes the following options:

- `-n name` will set the name of the build directory to the listed `name`. The default is `$NAME-$VERSION`. Other possibilities include `$NAME`, `${NAME}${VERSION}`, or whatever the main tar file uses. (Please note that these "$" variables are *not* real variables available within the spec file. They are really just used here in place of a sample name. You need to use the real name and version in your package, not a variable.)

- `-c` will create and cd to the named directory *before* doing the untar.
- `-b #` will untar Source# *before* cd'ing into the directory (and this makes no sense with `-c` so don't do it). This is only useful with multiple source files.
- `-a #` will untar Source# *after* cd'ing into the directory.
- `-T` This option overrides the default action of untarring the Source and requires a `-b 0` or `-a 0` to get the main source file untarred. You need this when there are secondary sources.
- `-D` Do *not* delete the directory before unpacking. This is only useful where you have more than one setup macro. It should *only* be used in setup macros *after* the first one (but never in the first one).

The next of the available macros is the `%patch` macro. This macro helps automate the process of applying patches to the sources. It takes several options, listed below:

- `#` will apply Patch# as the patch file.
- `-p #` specifies the number of directories to strip for the patch(1) command.
- `-P` The default action is to apply `Patch` (or `Patch0`). This flag inhibits the default action and will require a `0` to get the main source file untarred. This option is useful in a second (or later) `%patch` macro that required a different number than the first macro.
- You can also do `%patch#` instead of doing the real command: `%patch # -P`

That should be all the macros you need. After you have those right, you can also do any other setup you need to do via `sh` type scripting. Anything you include up until the `%build` macro (discussed in the next section) is executed via `sh`. Look at the example above for the types of things you might want to do here.

6.5 Build

There aren't really any macros for this section. You should just put any commands here that you would need to use to build the software once you had untarred the source, patched it, and cd'ed into the directory. This is just another set of commands passed to `sh`, so any legal `sh` commands can go here (including comments). **Your current working directory is reset in each of these sections to the toplevel of the source directory**, so keep that in mind. You can `cd` into subdirectories if necessary.

6.6 Install

There aren't really any macros here, either. You basically just want to put whatever commands here that are necessary to install. If you have `make install` available to you in the package you are building, put that here. If not, you can either patch the makefile for a `make install` and just do a `make install` here, or you can hand install them here with `sh` commands. You can consider your current directory to be the toplevel of the source directory.

6.7 Optional pre and post Install/Uninstall Scripts

You can put scripts in that get run before and after the installation and uninstallation of binary packages. A main reason for this is to do things like run `ldconfig` after installing or removing packages that contain shared libraries. The macros for each of the scripts is as follows:

- `%pre` is the macro to do pre-install scripts.
- `%post` is the macro to do post-install scripts.
- `%preun` is the macro to do pre-uninstall scripts.
- `%postun` is the macro to do post-uninstall scripts.

The contents of these sections should just be any `sh` style script, though you do *not* need the `#!/bin/sh`.

6.8 Files

This is the section where you *must* list the files for the binary package. RPM has no way to know what binaries get installed as a result of `make install`. There is *NO* way to do this. Some have suggested doing a `find` before and after the package install. With a multiuser system, this is unacceptable as other files may be created during a package building process that have nothing to do with the package itself.

There are some macros available to do some special things as well. They are listed and described here:

- `%doc` is used to mark documentation in the source package that you want installed in a binary install. The documents will be installed in `/usr/doc/$NAME-$VERSION-$RELEASE`. You can list multiple documents on the command line with this macro, or you can list them all separately using a macro for each of them.
- `%config` is used to mark configuration files in a package. This includes files like sendmail.cf, passwd, etc. If you later uninstall a package containing config files, any unchanged files will be removed and any changed files will get moved to their old name with a `.rpmsave` appended to the filename. You can list multiple files with this macro as well.
- `%dir` marks a single directory in a file list to be included as being owned by a package. By default, if you list a directory name *WITHOUT* a `%dir` macro, *EVERYTHING* in that directory is included in the file list and later installed as part of that package.
- `%files -f <filename>` will allow you to list your files in some arbitrary file within the build directory of the sources. This is nice in cases where you have a package that can build it's own filelist. You then just include that filelist here and you don't have to specifically list the files.

The biggest caveat in the file list is listing directories. If you list `/usr/bin` by accident, your binary package will contain *every* file in `/usr/bin` on your system.

6.9 Building It

6.9.1 The Source Directory Tree

The first thing you need is a properly configured build tree. This is configurable using the `/etc/rpmrc` file. Most people will just use `/usr/src`.

You may need to create the following directories to make a build tree:

- `BUILD` is the directory where all building occurs by RPM. You don't have to do your test building anywhere in particular, but this is where RPM will do it's building.
- `SOURCES` is the directory where you should put your original source tar files and your patches. This is where RPM will look by default.
- `SPECS` is the directory where all spec files should go.
- `RPMS` is where RPM will put all binary RPMs when built.
- `SRPMS` is where all source RPMs will be put.

6.9.2 Test Building

The first thing you'll probably want to to is get the source to build cleanly without using RPM. To do this, unpack the sources, and change the directory name to $NAME.orig. Then unpack the source again. Use this source to build from. Go into the source directory and follow the instructions to build it. If you have to edit things, you'll need a patch. Once you get it to build, clean the source directory. Make sure and remove any files that get made from a `configure` script. Then `cd` back out of the source directory to its parent. Then you'll do something like:

```
diff -uNr dirname.orig dirname > ../SOURCES/dirname-linux.patch
```

This will create a patch for you that you can use in your spec file. Note that the "linux" that you see in the patch name is just an identifier. You might want to use something more descriptive like "config" or "bugs" to describe *why* you had to make a patch. It's also a good idea to look at the patch file you are creating before using it to make sure no binaries were included by accident.

6.9.3 Generating the File List

Now that you have source that will build and you know how to do it, build it and install it. Look at the output of the install sequence and build your file list from that to use in the spec file. We usually build the spec file in parallel with all of these steps. You can create the initial one and fill in the easy parts, and then fill in the other steps as you go.

6.9.4 Building the Package with RPM

Once you have a spec file, you are ready to try and build your package. The most useful way to do it is with a command like the following:

```
rpm -ba foobar-1.0.spec
```

There are other options useful with the `-b` switch as well:

- `p` means just run the `prep` section of the specfile.
- `l` is a list check that does some checks on `%files`.
- `c` do a prep and compile. This is useful when you are unsure of whether your source will build at all. It seems useless because you might want to just keep playing with the source itself until it builds and then start using RPM, but once you become accustomed to using RPM you will find instances when you will use it.
- `i` do a prep, compile, and install.
- `b` prep, compile, install, and build a binary package only.
- `a` build it all (both source and binary packages).

There are several modifiers to the `-b` switch. They are as follows:

- `--short-circuit` will skip straight to a specified stage (can only be used with c and i).
- `--clean` removes the build tree when done.
- `--keep-temps` will keep all the temp files and scripts that were made in /tmp. You can actually see what files were created in /tmp using the `-v` option.
- `--test` does not execute any real stages, but does keep-temp.

6.10 Testing It

Once you have a source and binary rpm for your package, you need to test it. The easiest and best way is to use a totally different machine from the one you are building on to test. After all, you've just done a lot of `make install`'s on your own machine, so it should be installed fairly well.

You can do an `rpm -u packagename` on the package to test, but that can be deceiving because in building the package, you did a `make install`. If you left something out of your file list, it will not get uninstalled. You'll then reinstall the binary package and your system will be complete again, but your rpm still isn't. Make sure and keep in mind that just because you do a `rpm -ba package`, most people installing your package will just be doing the `rpm -i package`. Make sure you don't do anything in the `build` or `install` sections that will need to be done when the binaries are installed by themselves.

6.11 What to do with your new RPMs

Once you've made your own RPM of something (assuming its something that hasn't already been RPM'ed), you can contribute your work to others (also assuming you RPM'ed something freely distributable). To do so, you'll want to upload it to *ftp.redhat.com* (`ftp://ftp.redhat.com`).

6.12 What Now?

Please see the above sections on Testing and What to do with new RPMs. We want all the RPMs available we can get, and we want them to be good RPMs. Please take the time to test them well, and then take the time to upload them for everyone's benefit. Also, *please* make sure you are only uploading *freely available software*. Commercial software and shareware should *not* be uploaded unless they have a copyright expressly stating that this is allowed. This includes Netscape software, ssh, pgp, etc.

7 Multi-architectural RPM Building

RPM can now be used to build packages for the Intel i386, the Digital Alpha running Linux, and the Sparc. It has been reported to work on SGI's and HP workstations as well. There are several features that make building packages on all platforms easy. The first of these is the "optflags" directive in the `/etc/rpmrc`. It can be used to set flags used when building software to architecture specific values. Another feature is the "arch" macros in the spec file. They can be used to do different things depending on the architecture you are building on. Another feature is the "Exclude" directive in the header.

7.1 Sample spec File

The following is part of the spec file for the "fileutils" package. It is setup to build on both the Alpha and the Intel.

```
Summary: GNU File Utilities
Name: fileutils
Version: 3.16
Release: 1
Copyright: GPL
Group: Utilities/File
Source0: prep.ai.mit.edu:/pub/gnu/fileutils-3.16.tar.gz
Source1: DIR_COLORS
Patch: fileutils-3.16-mktime.patch

%description
These are the GNU file management utilities.  It includes programs
to copy, move, list, etc, files.

The ls program in this package now incorporates color ls!
```

```
%prep
%setup

%ifarch alpha
%patch -p1
autoconf
%endif
%build
configure --prefix=/usr --exec-prefix=/
make CFLAGS="$RPM_OPT_FLAGS" LDFLAGS=-s

%install
rm -f /usr/info/fileutils*
make install
gzip -9nf /usr/info/fileutils*

.
.
.
```

7.2 Optflags

In this example, you see how the "optflags" directive is used from the `/etc/rpmrc`. Depending on which architecture you are building on, the proper value is given to `RPM_OPT_FLAGS`. You must patch the Makefile for your package to use this variable in place of the normal directives you might use (like `-m486` and `-O2`). You can get a better feel for what needs to be done by installing this source package and then unpacking the source and examine the Makefile. Then look at the patch for the Makefile and see what changes must be made.

7.3 Macros

The `%ifarch` macro is very important to all of this. Most times you will need to make a patch or two that is specific to one architecture only. In this case, RPM will allow you to apply that patch to just one architecture only.

In the above example, fileutils has a patch for 64 bit machines. Obviously, this should only be applied on the Alpha at the moment. So, we add an `%ifarch` macro around the 64 bit patch like so:

```
%ifarch axp
%patch1 -p1
%endif
```

This will insure that the patch is not applied on any architecture except the alpha.

7.4 Excluding Architectures from Packages

So that you can maintain source RPMs in one directory for all platforms, we have implemented the ability to "exclude" packages from being built on certain architectures. This is so you can still do things like

```
rpm --rebuild /usr/src/SRPMS/*.rpm
```

and have the right packages build. If you haven't yet ported an application to a certain platform, all you have to do is add a line like:

```
ExcludeArch: axp
```

to the header of the spec file of the source package. Then rebuild the package on the platform that it does build on. You'll then have a source package that builds on an Intel and can easily be skipped on an Alpha.

7.5 Finishing Up

Using RPM to make multi-architectural packages is usually easier to do than getting the package itself to build both places. As more of the hard packages get built this is getting much easier, however. As always, the best help when you get stuck building an RPM is to look a similar source package.

8 Copyright Notice