RPM HOWTO

Donnie Barnes, djb@redhat.com

V1.2, August 30, 1995

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.

RPM can even provide an excellent method to upgrade an existing system. The database won't be as up to date as a machine that was completely installed with RPM, but it will still contain anything installed with RPM. It can also be used to package commercial software.

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 RHCL. 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 Commercial Linux. If you don't want to do that, you can still get and use RPM. It can be acquired from any Official Red Hat Mirror. Some of those are:

```
ftp.pht.com:/pub/linux/redhat
ftp.caldera.com:/pub/mirrors/redhat
sunsite.unc.edu:/pub/Linux/distributions/redhat
```

We are unsure at this point where to find it past there, but it will most likely just be a directory called RPM. We will make a tar file available with a README containing all the install instructions you should need.

3.2 RPM Requirements

The main requirement to run RPM is Perl 5.x. All of RPM is written in Perl. You must also have a working copy of `cpio` and `gunzip`, which most Linux distributions have now. While this system is intended for use with Linux, it may very well be portable to other Unix systems who meet the above conditions. 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 -u 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, however, that the current version of RPM will only do installs via FTP. You cannot run any of the more complex query options on packages at an FTP site.

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

```
rpm version 1.0
Copyright (C) 1995 - Red Hat Software
This may be freely redistributed under the terms of the GNU Public License

    Usage:
   --help               - print this message
    -q                  - query mode
      Package specification options:
        -a                - query all packages
        -f <file>+        - query package owning <file>
        -F                - like -f, but read file names from stdin
        -p <packagefile>+ - query (uninstalled) package <packagefile>
        -P                - like -f, but read package names from stdin
      Information selection options:
        -i                - display package information
        -l                - display package file list
        -s                - show file states (implies -l)
        -d                - list only documentation files (implies -l)
        -c                - list only configuration files (implies -l)

    -V
    -y
    --verify            - verify a package installation
                          same package specification options as -q

    --install <packagefile>
    -i <packagefile>    - install package
       -v               - be a little verbose
       -h
      --hash            - print hash marks as package installs (good with -v)
      --percent         - print percentages as package installs
```

```
  --force          - install despite potential conflicts
  --test           - don't install, but tell if it would work or not

--upgrade <packagefile>
-U <packagefile>    - upgrade package (same options as --install)

--uninstall <package>
-u <package>        - uninstall package

-b<stage> <spec>    - build package, where <stage> is one of:
                      p - prep (unpack sources and apply patches)
                      l - list check (do some cursory checks on %files)
                      c - compile (prep and compile)
                      i - install (prep, compile, install)
                      b - binary package (prep, compile, install, package)
                      a - bin/src package (prep, compile, install, package)
  --short-circuit   - skip straight to specified stage (only for c,i)
  --clean           - remove build tree when done
  --keep-temps      - do not delete scripts (or any temp files) in /tmp
  --test            - do not execute any stages, implies --keep-temps
                      in /tmp - useful for testing
  --time-check <s>  - set the time check to S seconds (0 disables it)
```

First, I'll go through a synopsis of what all the options mean (don't worry, there may be alot of options, but we tried to make them all as intuitive as possible).

Options are nested, so the possible options are many. Here's a description in parallel with the `Usage` message:

- `help` prints the usage message
- `-q` is the query option. In its simplest form, you can do `rpm -q foobar` which would return `foobar-1.0-1`. (1.0 is the version number, 1 is the release number.)
- Several options may be used with `-q`:
  - `-a` will query all currently installed packages.
  - `-f <file>` will query the package owning <file>.
  - `-F` is the same as `-f` except you can give it filenames via stdin (ie. `ls /usr/bin | rpm -qF`).
  - `-p <packagefile>` will query the package. It is really only useful when combined with one of the Information Selection Options below.
  - `-P` is like -p, except it takes its package filenames from stdin (ie. `ls /mnt/redhat/redhat-2.0/RPMS | rpm -qP`).
- Several Information Selection Options can be used with any combination of the above options. If none is given, the package name only is displayed.
  - `-i` displays package information such as Name, Description, Release, etc.

  - `-l` will display the file list from the entire package (all files that get installed). You can also use a `-v` with this to make the file list much more verbose.
  - `-s` shows you the state of all the files in the package. There are only two possible states, normal and missing.
  - `-d` outputs a list of just the files marked as documentation (man pages, info pages, README's, etc). `-v` will give even more info.
  - `-c` outputs a list of only the configuration files (sendmail.cf, passwd, inittab, etc.) `-v` will give more info about the files.

- {`-V`,`-y`,`--verify`} are the verify options. All are interchangeable. They all take the same Package Specification and Information Selection options as the `-q` option. I'll list some examples:
  - To verify a package containing particular file, do:

    ```
    rpm -yf /bin/vi
    ```

  - To verify ALL your files, do:

    ```
    rpm -ya
    ```

  - To verify files on your system versus the files in a .rpm file, do:

    ```
    rpm -Vp foobar-1.0-1.rpm
    ```

- `--install` installs an rpm file.
- `-i` installs an rpm file.
  - `--hash, -h` is a very cool option for watching the package install (much like 'hash' in ftp).
  - `--percent` prints the percentages as a package installs (but is only useful for interfacing with other tools...is not really human readable).
  - `--force` will force an install of a binary package even though it may already exist in the database.
  - `--test` will tell you if installing would work or not (do you have a conflict with an already installed package).
- `-U` upgrades a package. This option installs the new package and then uninstalls the old one without hurting the new one...
- `-v` be verbose in the output of what's going on.
- `-vv` be *very* verbose in the output of what's going on.
- `--uninstall, -u` <package> to uninstall a package
- `-b` to build a package (from sources and a spec file). This option will be discussed more at length in the next section, Building RPMs.

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-1.0-1.i386.rpm
  ```

  The output would be:

  ```
  Name        : koules                     Distribution: RHCL 2.0
  Version     : 1.0                              Vendor: Red Hat Software
  Release     : 1                            Build date: Tue Aug 29 12:53:21 1995
  Install date: <not installed>              Build host: daffy.redhat.com
  Group       : Games
  Size        : 403105
  Description : well done SVGAlib game
  ```

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

  ```
  rpm -qpl koules-1.0-1.i386.rpm
  ```

  The output is:

  ```
  /usr/man/man6/koules.6
  /usr/lib/games/kouleslib/start.raw
  /usr/lib/games/kouleslib/end.raw
  /usr/lib/games/kouleslib/destroy2.raw
  /usr/lib/games/kouleslib/destroy1.raw
  /usr/lib/games/kouleslib/creator2.raw
  /usr/lib/games/kouleslib/creator1.raw
  /usr/lib/games/kouleslib/colize.raw
  /usr/lib/games/kouleslib
  /usr/games/koules
  ```

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.

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
require_distribution: 1
require_group: 1
distribution: RHCL 2.0
vendor: Red Hat Software
arch_sensitive: 1
```

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).

The last line is `arch_sensitive`. This specifies where the binary RPM's go and what they are named. Right now, only i386 is defined as a type within RPM. That means if you are building on an Intel machine and have this value set to true, your RPM's will go in `/usr/src/redhat-2.0/RPMS/i386/` and their name will be something like `foobar-1.0-1.i386.rpm`. If you set this value to `0`, the RPM's will be placed in `/usr/src/redhat-2.0/RPMS/` and will be named something like `foobar-1.0-1.bin.rpm`. This does not affect the name or placement of the source RPM, however.

Future versions of RPM will have some macros to allow for the use of the same source file and different patches depending on architecture. This is unimplemented as of version 1.0.

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

- `topdir` to specify the top level directory for building. In Red Hat 2.0, this directory is `/usr/src/redhat-2.0`.
- `specdir` is the directory under topdir to use for the spec files. The default for this is `SPECS`.
- `builddir` specifies the top level of the build directory. The default for this is `BUILD`.
- `sourcedir` specifies the top level of the source directory. The defualt for this is `SOURCES`. This is where the pristine tar files, the patches, and the icons go.
- `rpmdir` sets the directory for the binary RPMs. The default for this is `RPMS`.

- `srcrpmdir` sets the directory for the source RPMs. The default for this is `SRPMS`.
- `docdir` specifies where the documentation should be installed. By default, this is `/usr/doc`.
- `libdir` sets the path for the RPM database. By default, this is `/var/lib/rpm`.
- `timecheck` sets whether or not to do a timecheck by default.

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):

```
Description: VIsual editor iMproved
Name: vim
Version: 3.0
Release: 1
Icon: vim.gif
Source: sunsite.unc.edu:/pub/Linux/apps/editors/vi/vim-3.0.tar.gz
Patch: vim-3.0-make.patch
Copyright: distributable
Group: Applications/Editors

%prep
%setup
%patch -p1
cd src
cp makefile.unix makefile

%build
cd src
make

%install
rm -f /bin/vim
cd src
make install
ln -sf vim /bin/vi

%files
%doc doc/reference.doc doc/unix.doc tutor/tutor
/bin/vim
/bin/vi
```

```
/usr/man/man1/vim.1
/usr/lib/vim.hlp
```

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:

- `Description:` This one is kind of obvious. You can span multiple lines by ending each line with a backslash.
- `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 must like the 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.
- `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
```

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.
- `-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.

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. Right now, you must use:

```
/usr/src/redhat-2.0
```

as the place to build. This will be configurable in later versions and the advanced user (and perl programmer) might choose to edit the source to put everything somewhere else for now.

You don't need to create any other directories, but several will be created for you. These are:

- `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`. 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 -v 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.

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.
- `--time-check #` is very useful. By default, the time-check value is 7200 seconds (two hours). What this does is check all the files in `%files` and warns you if they are more than `#` seconds old (or the default). This lets you make sure that the newly created binaries are getting installed and not old ones that just happen to be still lying around. This author can attest to the value of this feature after having to release several RPP updates because old binaries were accidentally included. You can also turn this off (useful when building binary only packages of commercial software) by setting the value to zero.

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 an FTP site somewhere. We hope RPM will become a standard that everyone starts using. If that is the case, you should probably upload your RPMs to sunsite.unc.edu. Until then, please upload them to our official Red Hat Mirror, ftp.pht.com:/pub/linux/redhat/Incoming. We are currently mirrored on several other sites, and this is the best place to find new RPMs.

7 Advanced RPM Building

RPM has some very advanced features available for larger, more complex packages. It has the ability to build and output multiple binary subpackages. An example of this is the ability to produce separate Tcl/Tk binary packages from one spec file. Another example is the ability to use one spec file to create a single XFree86 package with no servers, and a separate package for each of the servers.

7.1 How to Get Started

The best way to get started is to look at an example spec file. The following tcl/tk spec file is a good one to start with (though you can also view the spec file of any package by installing the sources and looking in /usr/src/redhat-2.0/SPECS):

```
%package tcl
Description: Tool Command Language
Name: tcltk
Version: 7.4_4.0
Release: 1
Icon: tcl.gif
Source0: ftp.cs.berkeley.com:/pub/tcl/tcl7.4.tar.Z
Source1: ftp.cs.berkeley.com:/pub/tcl/tk4.0.tar.Z
Copyright: BSD
Group: Development/Languages/Tcl
Patch0: sunsite.unc.edu:/pub/Linux/devel/tcl7.4-1.diff.gz
Patch1: sunsite.unc.edu:/pub/Linux/devel/tk4.0-1.diff.gz
%package tk
Icon: tk.gif
Description: Tk toolkit
Group: Development/Languages/Tcl

%prep
%setup -T -c -a 0
%setup -T -D -a 1
%patch0 -p0
%patch1 -p0

%build
cd tcl7.4
./configure --prefix=/usr
make
cd ../tk4.0
./configure --prefix=/usr
make

%install
cd tcl7.4
make install
ln -sf libtcl7.4.a /usr/lib/libtcl.a
ln -sf libtcl7.4.so.1 /usr/lib/libtcl.so.1
ln -sf libtk4.0.a /usr/lib/libtk.a
ln -sf libtk4.0.so.1 /usr/lib/libtk.so.1
cd ../tk4.0
make install

%post tcl
```

```
/sbin/ldconfig

%post tk
/sbin/ldconfig

%postun tcl
/sbin/ldconfig

%postun tk
/sbin/ldconfig

%files tcl
/usr/lib/libtcl7.4.a
/usr/lib/libtcl.a
/usr/lib/libtcl7.4.so.1
/usr/lib/libtcl.so.1
/usr/include/tcl/tcl*
/usr/bin/tclsh
/usr/bin/tclsh7.4
/usr/lib/tcl7.4
/usr/man/man1/*tcl
/usr/man/man3/*tcl

%files tk
/usr/lib/libtk4.0.a
/usr/lib/libtk.a
/usr/lib/libtk4.0.so.1
/usr/lib/libtk.so.1
/usr/include/tcl/ks_names.h
/usr/include/tcl/default.h
/usr/include/tcl/tk*
/usr/lib/tk4.0
/usr/man/man1/*tk
/usr/man/man3/*tk
/usr/bin/wish
/usr/bin/wish4.0
```

7.2 Sub-Packages

One of the main advanced features of RPM is the ability to build subpackages. They are easy to build as for most macros you can just add the subpackage name as a parameter for anything specific to a subpackage (and if you leave it off the section will apply to the main package).

7.3 The Header

The header only has one major difference, the `%package` macro. This macro is used in the header to tell which subpackage name to match the description with. If you omit the macro in the initial part of the header, you will get a main package with no change to the name. In the XFree86 package, however, there is no `%package` macro in the top of the header. This is because we wanted a base XFree86 package with all the common stuff in it and then several subpackages (XFree86-SVGA, etc.) with the servers. Tcl/Tk does not need a main package, so the macro is at the top.

Another difference is the fact that this package has multiple source and patch lines. If you'll notice, there is now a `Source0` line instead of just `Source`. They are functionally equivalent, though it is a good idea to use `Source0` when there is more than one source file (and the same applies to patches as well).

7.4 Prep

Prep is basically the same as in the simple example, except it uses more of the options available to the setup and patch macros.

7.5 Build

Build is basically the same, with the exception that the setup macro above used the `-T` option. Because of that, you have to do a manual `cd` to get into the source directory.

You will also notice that the build does a `configure` before it can build. This is the section where any of this type of configuration should go.

7.6 Install

Again, everything is pretty normal with the exception of the fact that you must manually `cd` into the source directories.

7.7 Optional pre and post Install/Uninstall Scripts

This section is almost the same as in a simple RPM case (see the above section). It has two post install scripts that run ldconfig for each of the subpackages upon install. It *should* have two post uninstall scripts to run ldconfig as well.

7.8 Files

Here you will declare which files go in which packages. You really have multiple file sections, each started with a new `%files` macro and the name of the subpackage (except in the case where you have a main package...that `%files` macro will have no argument given to it). The other macros (doc, config, etc) work exactly the same as in the simple case.

You also have the option to use the * to glob filenames out of a directory. You need to be careful with this (perhaps test it first) so as not to include files you didn't mean to. The above example does this with the man pages.

7.9 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.

8 Copyright Notice