The RPM Packaging System

Marc Ewing
Erik Troan
Red Hat Software

ABSTRACT

With the advent of a wide range of freely available operating systems, the need for a freely available packaging system has arisen. The standard packaging options such as System V's pkgadd suite and the IEEE standard for system administration (IEEE P1387.2) don't address many of the key concepts necessary for a freely available packaging standard.

Such a system must encompass source packages as well as binary packages. It must allow for automatic, reproducible construction of binary packages from source packages. In addition the system should support querying of the installed package system state, verification of installed packages, and must perform intelligent package upgrading with due consideration for user-modifiable "configuration" files.

Since many of these freely available operating systems run on multiple hardware architectures, the packaging system should support multiple architectures. Ideally, the system would allow for a single source package from which it can generate binary packages for each supported architecture.

The Red Hat package Manager (RPM) (distributed by Red Hat Software under the terms of the GPL) is an attempt to address these issues. It supports source packages which consist of original "pristine" source code as distributed by the author, zero or more patches that were required to build the software, and instructions for building the software. Many of the problems of multiple architectures have been addressed by a simple conditional building mechanism. Indeed, using RPM, Red Hat Software is able to maintain a single collection of source packages that is used to build Red Hat Linux for iX86 and Alpha platforms.

The original version of RPM was implemented in perl to enable rapid deployment. This version of RPM has been used under Linux on multiple platforms (including Intel 80x86 and Digital Equipment Corporation's Alpha processors) as well as various versions of Irix, AIX, and Digital Unix. The second version is a complete rewrite in C (primarily for performance reasons) and includes significant improvements in both the package and database formats while retaining full backward compatibility.

1 Design Goals and Introduction

The most important design goals are features of RPM are:

Upgradability With RPM you can upgrade individual components of your system without completely reinstalling. When you get a new release of a distribution based on RPM (such as Red Hat Commercial Linux), you don't need to reinstall your machine (as you do with distributions based on other packaging systems). You instead run an upgrade script which checks to see what packages you have installed, and upgrades those as necessary. Individual configuration files in packages are preserved across upgrades, so you won't lose your customizations.

Powerful Querying You can search the RPM database for packages or specific files. It is easy to find out what package a file belongs to. The RPM package files themselves are compressed archives, but you can query them easily and quickly because a binary header is added to the compressed archive and contains all of the information on the package in an uncompressed format.

System Verification Another feature is verification of packages. If you are worried that you deleted a file important for some package, just verify the package. You will be notified of any anomalies. If you find any, you can reinstall the package if necessary. Any configuration files that you modified are preserved during reinstallation.

Pristine Sources A crucial design goal for RPM was to allow the use of “pristine” software sources, as distributed by the original authors of the software. With RPM, you have the pristine sources along with any patches that we used to compile them. This is a big advantage for 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. You can look at the patch to see what you *might* need to do. All the compiled defaults, and all of changes that were made to get the software to build properly on Linux are easily visible this way.

1.1 Building Packages: The Spec File

The most important thing to understand, from a package builder’s perspective, is the “spec” file. The spec file holds miscellaneous package information, specifies all sources and patches, contains shell commands to prepare, build, and install from the sources, and a list of files the binary package should contain. The basic sections of the spec file are:

Preamble section contains information about the package, including the name, version, release, description, copyright, etc.

```
Name: textutils
Version: 1.10
Release: 1
Description: utilities to manipulate ...
...
```

Prep section contains shell commands (and special macros) to unpack and prepare the sources for compilation.

```
%prep
%setup
```

Build section contains shell commands to compile the sources.

```
%build
./configure --prefix=/usr
make
```

Install section contains shell commands to install the compiled programs

```
%install
make install
```

Files section is a simple list of files to be packaged.

```
%files
/usr/bin/wc
...
```

2 Source and Patch Management

2.1 Concept

In addition to the expected user-side features of a packaging system, a packaging system for managing free software demands special features to aide the developer who must maintain dozens of packages whose sources come from many different authors, and whose installation needs to be massaged to conform to a single file system structure.

One of the most important design decisions was the decision to use "pristine" sources. These are the sources exactly as distributed by the original author of the software, typically a gzipped tar file. It is very important that the automatic build process start with the pristine sources. This ensures that the builds are repeatable and that we have a clearly defined division between author-supplied sources, and Red Hat (or other third party) modifications.

This division also eases the job of maintaining our source tree. When the author releases a new set of sources, generating a new package is often as simple as replacing the old pristine source tar file with the new version and changing the version numbers in the spec file. Having our patches separate from the pristine sources also makes it easy to forward the patches on to the author for possible inclusion in the next release. If the author incorporates the patches the spec file is easily remove the lines from the spec file that apply the patch.

The original "pristine sources plus patches and build instructions" idea was conceived by Kevin Martin, Rik Faith, and Doug Hoffman. They implemented it in their PMS packaging system which is used by the BOGUS Linux distribution.

2.2 Implementation

In the "preamble" section of the spec file, the sources and patches are specified like so:

```
Source: ftp://prep.ai.mit.edu/pub/gnu/textutils-1.10.tar.gz
Patch: textutils-1.10-misc.patch
```

The "ftp://prep.ai.mit.edu/pub/gnu/" part of the source line is actually ignored by RPM, but it is recommended that you always specify the complete URL to the sources so you can get the sources again, or check for new releases.

Each source and each patch is numbered for later reference in the prep section of the spec file. In this case they are each implicitly assigned the number 0, and are equivalent to:

```
Source0: ftp://prep.ai.mit.edu/pub/gnu/textutils-1.10.tar.gz
Patch0: textutils-1.10-misc.patch
```

If your package requires multiple sources (or patches) you must assign unique numbers to each source (or patch). For example:

```
Source0: ftp://prep.ai.mit.edu/pub/gnu/textutils-1.10.tar.gz
Source1: ftp://ftp.redhat.com/misc/textutils-help.tar.gz
Patch0: textutils-1.10-misc.patch
Patch1: textutils-1.10-fsstnd.patch
```

In the prep section you write commands to unpack the sources and apply the patches:

```
%prep
zcat /usr/src/SOURCES/textutils-1.10.tar.gz | tar -xvf-
cd textutils-1.10
patch -p1 < /usr/src/SOURCES/textutils-1.10-misc.patch
```

Since most of these unpacking and patching operations are very common, two "macros" are provided to make things simpler. The above commands could be rewritten as:

```
%prep
%setup
%patch -p1
```

By default, the "`%setup`" macro unpacks the source numbered 0, and similarly the "`%patch`" macro applies patch number 0. If there were multiple sources and patches (as in example listed above with 2 sources and 2 patches) we might write:

```
%prep
%setup
%setup -D -T -b 1
%patch -p1
%patch -P 1 -p1
```

A quick explanation of the options to `%setup`:

-D	do not remove existing unpacked sources before unpacking
-T	inhibit unpacking source 0 (the default)
-b 1	unpack source 1 before changing into the root of the sources

and to `%patch`:

-P	inhibit applying patch 0 (the default)
1	apply patch 1
-p1	pass "-p1" option to the patch program

These macros take care of uncompressing the sources/patches with the appropriate program, and have options to handle cases where the sources do not unpack into a directory named NAME-VERSION. For complete option details, see the RPM man page.

The macros can be intermixed with shell commands:

```
%prep
%setup
rm -f textutils-1.10/Makefile
%patch -p1
```

3 Building Packages

3.1 The Files Section

We have already seen how to construct the spec file preamble, prep, build, and install sections. The final section is the `%files` section. The files section of the spec file for bash is listed below:

```
%files
%doc NEWS README RELEASE
%config /etc/bashrc
/bin/bash
/usr/info/bash.info.gz
/usr/man/man1/bash.1
/usr/bin/bashbug
```

Note the two tags `%doc` and `%config`. The `%config` tag marks `/etc/bashrc` as a configuration file, which is specially handled when upgrading. The `%doc` tag has two modes. The first mode (which is not used above) marks a file as a documentation file. When `%doc` tag is used in front of a file name with a leading "/" it signifies this mode. This marking is simply for the user's convenience in locating documentation on the system (in fact, you could build a documentation browser that queries the RPM database to find documentation files). The other mode, which is used in the above spec file fragment, specifies files in the package sources that should be installed in `/usr/doc/<packagename>-<version>-<release>`. This is a convenience to package builders, as many times there are important documentation files in the sources which are not installed with a `make install`.

3.2 Building the Source and Binary Packages

The following options direct RPM to build a package from a spec file given on the command line:

Option	Stages Executed
`-bp`	prep
`-bc`	prep, build
`-bi`	prep, build, install
`-bb`	prep, build, install, binary packaging
`-ba`	prep, build, install, binary and source packaging

For instance:

```
rpm -ba fileutils-3.12.spec
```

will do all stages of compilation and will construct the binary and source packages. Note that each of these options always start from the very beginning: the prep stage. This ensures that the packages are always built starting with the original pristine sources. However, during development of a package (ie, while constructing the spec file and testing) it can be quite a burden to start from the prep stage each time, especially as the prep stage usually removes any existing sources (through the `%setup` macro). To make life easier, you can use `--short-circuit` with the `-bc` and `-bi` options to jump straight to the last stage of each. Note that `--short-circuit` does *not* work with either `-bb` or `-ba`, again, to ensure that resultant packages are always built starting from the original pristine sources.

4 Querying and Verifying

Querying the database of installed packages is accomplished with `rpm -q`. A simple use is `rpm -q foobar` which will print the package, version, and release number of the installed foobar package (if it is installed): `foobar-1.0-1`.

Instead of specifying the package name, you can use the following options with `-q` to specify what package(s) you want to query. These are called *Package Specification Options*.

- `-a` queries all currently installed packages.
- `-f` <`file`> will query the package owning <file>.
- `-F` is the same as `-f` except it takes filenames via stdin (ie. `find /usr/bin | rpm -qF`).
- `-p` <`packagefile`> queries the package <packagefile>.
- `-P` is like `-p` except it takes package filenames from stdin (ie. `find /mnt/redhat/redhat-2.0/RPMS | rpm -qP`).

There are a number of ways to specify what information to display about queried packages. The following options are used to select the information you are interested in. These are called *Information Selection Options*.

- `-i` displays package information such as name, description, release, size, build date, install date, vendor, and other miscellaneous information.
- `-l` displays the file list for the package (all files that get installed).
- `-s` displays the state of all the files in the package. There are only two possible states, normal and missing.
- `-d` displays a list of files marked as documentation (man pages, info pages, README's, etc).

- `-c` displays a list of files marked as configuration files. These are the files you change after installation to adapt the package to your system (sendmail.cf, passwd, inittab, etc).

Package verification involves checking the owner, group, permissions, size, MD5 sum, major and minor numbers, symlink, and mtime of each installed file in a package. `rpm -V` verifies a package. You can use any of the *Package Selection Options* listed for querying to specify the packages you wish to verify. A simple use is `rpm -V foobar` which verifies that all the files in the foobar package are as they were when they were originally installed. For example:

- To verify a package containing particular file:

  ```
  rpm -Vf /bin/vi
  ```

- To verify ALL installed packages:

  ```
  rpm -Va
  ```

- To verify an installed package against an RPM package file:

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

  This can be useful if you suspect that your RPM databases are corrupt.

The original RPM checked all of the file attributes during verification. This proved to be a problem for configuration files and devices. For example the owner of /dev/ttyS0 is expected to change, so verifying that attribute makes little sense. RPM 2.0 added the `%verify` option which the packager uses in the `%files` section of the spec file to control the verification of individual files. Its syntax is as follows:

```
%verify([except] [owner|group|mode|md5|size|maj|min|symlink|mtime]*) /file..
```

The developer lists the aspects of the file that RPM should verify inside the parentheses. If the `except` keyword is listed first, then RPM will verify all aspects *except* those listed.

5 Upgrade Issues

Some packages contain files that the user is meant to modify. For instance, the sendmail package contains `/etc/sendmail.cf`. In many cases a user may need to modify this file to properly configure mail delivery. When a new release of the sendmail package is made, and the user upgrades, those changes need to be preserved.

Three features are used to enable such "smart" upgrading:

MD5 Checksums Each file in a package has an associated MD5 checksum, which is a unique digital signature for the content of that file. If the file changes as much as a single character, the MD5 checksum will change.

%config Files that are to be handled specially across upgrades (eg, `/etc/sendmail.cf`) are designated as "configuration files" by prefixing them with `%config` in the `%files` section of the spec file.

File States In the RPM database, files are marked as one of the following:

> **normal** The file has not been overwritten by any other package.
>
> **shared** The file is also "owned" by another package, but the MD5 checksums are the same.
>
> **overwritten** The file has been overwritten by a file from another package with a different MD5 checksum.

With the file state flags, it is possible to have two versions of the same package installed at the same time. When one version is uninstalled the shared and overwritten files are *not* removed, as they are owned by some other package that will still be installed.

To upgrade a package, RPM first installs the new version of the package, with one important difference from a normal install. For files marked as configuration files, RPM takes the following actions:

A	B	C	Action
x	x	x	install new
x	x	y	install new
x	y	x	leave existing
x	y	y	install new
x	y	z	save existing, install new

Where:

A	original MD5 sum of file in old package
B	MD5 sum of file on disk
C	MD5 sum of file in new package

To "save" an existing file, it is copied to `file.rpmsave`. After this, RPM simply uninstalls the old package using normal methods.

In plain English, configuration files are handled like this: If the file does not change from one package version to the next, it is safe to leave alone whatever is on disk. If the file changes from one version to the next, and the file on disk has not been changed, then it is safe to install the new file. And finally, if the file changes from one version to the next, and the file on disk has changed, the it must be saved to `file.rpmsave` before installing the new file.

If any configuration files are "saved" the user is given a warning. The user should then reconcile the changes he or she made with the new configuration file.

6 Subpackages

Often it is desirable to generate more than one package from a single set of sources and a single spec file. For instance, when we built the ncurses package, we split it into two parts: one contains the terminfo library and the ncurses shared libraries, and a second "developer" part contains static and debugging libraries, header files, and man pages. The rationale here is that the majority of users only need the minimum required to run binaries linked with ncurses. By splitting up ncurses into two packages we save those users over a 1MB of space on their hard drives.

To accomplish this, we use the `%package` directive in the spec file:

```
%package devel
Description: development libraries and man pages for ncurses
```

We also need to specify a file list:

```
%files devel
...
```

Running `rpm -ba` on this spec file results in two packages: ncurses-1.9.6-2.i386.rpm, and ncurses-devel-1.9.6-2.i386.rpm.

Like `%files`, the `%prein`, `%postin`, `%preun`, and `%postun` directives can also be "parameterized" for the additional packages.

For an example of a spec file that generates multiple packages, see the RPM-HOWTO.

7 Pre and Post Install (Uninstall) Operations

Oftentimes it is desirable to be able to run a program after (or before) a package is installed (or uninstalled). To facilitate this, RPM supports four additional sections in the spec file:

```
%prein
%postin
%preun
%postun
```

Like the `%build` and `%install` sections, you write simple shell commands in these sections, and they will be executed at the appropriate times.

8 Multi-Architecture Issues

As was our intent, Red Hat Linux is available on multiple platforms. We wanted RPM to have features that would help us keep our sanity while maintaining hundreds of packages on different architectures. It is important that the binary packages for each architecture are built from a single common source package, using the same spec file and the same available sources and patches. This ensures that our releases for each architecture are in sync with each other, and they are all built the same way.

The simple addition of "`%ifarch`" to the spec file syntax allows for conditional patching and building. For example, currently Linux only supports shared libraries on the Intel platform. The following conditionally applies a patch that generates the shared libs:

```
%ifarch i386
%patch 1 -P
%endif
```

You can list more than one architecture after the "`%ifarch`", and the test will succeed for any of the architectures.

The "`%ifarch`", optional "`%else`", and "`%endif`" can be used anywhere in the spec file, and can even cross preamble/prep/build/install/file section boundaries. Additional conditional expressions are "`%ifnarch`", "`%ifos`", and "`%ifnos`".

Sometimes, a package is only meant to be built on a single platform. For example, our Alpha and Intel releases each have their own C library, neither of which can be built on the other platform. Surrounding the entire spec file with an "`%ifarch`" is unsightly and the proper behavior for such a spec file (which is effectively empty) is not clear. For these cases, the packager can use "`Exclude: <arch>`" and/or "`Exclusive: <arch>`" lines in the preamble of the spec file.

The final feature related to compiler optimization flags. Before we had a product on the Alpha, we had a lot of build sections like:

```
%build
make -m486 -O2
```

This would not work on the Alpha. Now we would write:

```
%build
make $RPM_OPT_FLAGS
```

When building, RPM sets the `RPM_OPT_FLAGS` environment variable to a value listed in the `rpmrc` file (described below) for the current architecture.

With these features, it is quite easy to create packages that build cleanly on multiple architectures. However, there are stylistic concerns. For instance, it is good practice to create your patches such that you can apply them in all cases, using "`#ifdef`" to conditionalize on the architecutre, or perhaps on a variable set on the make command line.

9 The RPMRC File

The RPM system has many user configurable options and variables, many of which are set once and remain unchanged from invocation to invocation. So the user doesn't have to specify these each time it is run, RPM uses an "profile" file `/etc/rpmrc or $HOME/.rpmrc` to set a number of options and variables.

A sample configuration file looks like:

```
require_vendor: 1
require_distribution: 1
require_group: 1
distribution: RHCL 2.1
vendor: Red Hat Software
arch_sensitive: 1
topdir: /usr/src/redhat-2.1

optflags: i386 -O2 -m486
optflags: axp -O

search: /mnt/redhat/RedHat/RPMS
search: /usr/rhs/ftp/pub/redhat-2.1/updates/RPMS
search: ftp://ftp.redhat.com/pub/redhat-2.1/updates/RPMS
```

- The `require_vendor` line causes `rpm` to complain and stop if it can not determine a vendor. This can come from a vendor line in your `/etc/rpmrc` or from the rpm spec file itself. To turn this off, change the number to `0`. The same holds true for the `require_distribution` line.
- The `require_group` line causes `rpm` to complain and stop if there is no group line in the packages spec file.
- The `distribution` and `vendor` lines can be used to specify the product and the vendor that the package belongs to. These can also be specified in the RPM 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 `arch_sensitive` line specifies where the binary RPM's are placed and what they are named. For instance, if you are building on an Intel box, and have `arch_sensitive` set to `1`, your RPM's will go in *rpmdir*/i386 and their name will be something like `foobar-1.0-1.i386.rpm`. If you set `arch_sensitive` to `0`, the RPM's will be placed in *rpmdir* and will be named something like `foobar-1.0-1.bin.rpm`. This option does not affect the name or placement of the source RPM, however.
- The `search` lines are used to define a set of search paths for RPM packages (see below).

The following table is a complete list of options for RPM version 1.4.1 (the list for RPM 2.0 is not yet finalized, but will be a superset of this list). Default values are in [brackets].

`topdir` Specify the top level directory for building RPMS [`/usr/src`]. This is used to set paths for the the other build directories if none are specified.

`specdir` Directory to place spec files [`SPECS`].

`builddir` Directory where sources are unpacked and built [`BUILD`].

`sourcedir` Directory where sources and patches are placed when a source package is installed [`SOURCES`].

`rpmdir` Directory where binary RPMS are placed when built [`RPMS`].

`srcrpmdir` Directory where source RPMS are placed when built [`SRPMS`].

`docdir` Directory where documentation is placed when the `%doc` macro is used in a spec file [`/usr/doc`].

`libdir` Directory where the RPM databases are kept [`/var/lib/rpm`].

`timecheck` Specified in seconds. If any files in a package are older than this value (if non-zero), you will get a warning [`7200`].

`force` If non-zero, then `rpm` acts is if `--force` was specified [`0`].

`arch_sensitive` Controls placement and naming of binary packages [`0`].

`require_vendor` If non-zero, then a build will fail if `rpm` can not determine a vendor [`0`].

`require_distribution` If non-zero, then a build will fail if `rpm` can not determine a distribution [`0`].

`require_group` If non-zero, then a build will fail if there is no group specification in the spec file [`0`].

`require_icon` If non-zero, then a build will fail if there is no icon defined in the spec file [`0`].

`distribution` Defines a system-wide distribution in lieu of a distribution line in a spec file [*null*].

`vendor` Defines a system-wide vendor in lieu of a vendor line in a spec file [*null*].

`optflags` Defines the value to give the RPM_OPT_FLAGS environment variable when running build scripts on the specified platform [*null*].

`messagelevel` Controls the default "verbosity". 0 is quiet, 1 is normal, 2 is verbose, and 3 is a debug level [`1`].

10 File Formats

The format of both package files and the RPM database changed substantially. Between the 1.x versions of RPM and RPM 2.0, although RPM 2.0 transparently handles older package files and can convert old databases to the new format with no loss of information. This discussion describes the new format of RPM files.

Early versions of RPM (then called PM) used a gzipped file archive as the package format. We moved away from this for a couple of reasons. First, there was no simple way to imbed information like a description, file list, or icon. Secondly, querying was very slow. The time it took to uncompress the initial portion of the archive to retrieve information about the package was very noticeable when the system was scanning two hundred package from a CD-ROM.

Each package starts with a small fixed length "lead". The lead contains information specifying the type of package, a file format version, the package name, version, and release. This info can be used by the file(1) command to identify RPM packages even when they are renamed.

The next section is optional, and contains a digital signature. We plan to include support in RPM so that package builders can digitally "sign" (using PGP or a similiar mechanism) their packages, so users of the package can verify the integrity and/or origin of the package. This signature covers the third and fourth sections.

The third section is called the "header". The header is an extensible collection of information about the package, including the name, version, release, decsription, file names and MD5 sums, and other information. All numbers in the headers are stored using the standard network byte ordering to allow packages to be queried on all architectures.

The fourth and final section is a gzipped cpio archive of the files the package contains. This is the "meat" of the package. There are several reasons we decided to use cpio format archives instead of the more common tar format:

- tar will not overwrite symbolic links which are already present

- tar follows symbolic links when extracting an archive
- the standard tar command provides no method to include a directory without including the contents of that directory

While cpio has some drawbacks of its own, those are easily overcome by carefull handling of the archive within RPM itself.

The RPM database is built around a simple dynamic allocation scheme and indexed using Berekley db hash indexes. Berkeley db was chosed over gdbm due to its byte order independence. The hash tables index a file which contains the package headers and a minor amount of extra bookkeeping information.

11 Using RPM Effectively

RPM is a very useful tool for managing your system and diagnosing and fixing problems The best way to make sense of all the options is to look at some 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 enter:

  ```
  rpm -Va
  ```

  If any of the packages are listed as "bad" you may wish to verify them individually with `-v` for more detailed information:

  ```
  rpm -Vv foobar
  ```

  If some files are missing, or appear to have been corrupted, you should probably either re-install the package or uninstall, then re-install the package.

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

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

  The output would look like:

  ```
  xjewel-1.6-1
  ```

- We can combine the above two examples in the following scenario. Say you are having problems with `/usr/bin/paste`. You would like to verify the package that owns that program but you don't know which package that is. Simply enter:

  ```
  rpm -Vf /usr/bin/paste
  ```

  and the appropriate package will be verified.

- If you are using a program and want to find out more information about it, you can enter the following to find out what documentation came with the package that "owns" that program (in this case `ispell`):

  ```
  rpm -qdf /usr/bin/ispell
  ```

  The output would be:

  ```
  /usr/man/man4/ispell.4
  /usr/man/man4/english.4
  /usr/man/man1/unsq.1
  /usr/man/man1/tryaffix.1
  /usr/man/man1/sq.1
  /usr/man/man1/munchlist.1
  ```

```
/usr/man/man1/ispell.1
/usr/man/man1/findaffix.1
/usr/man/man1/buildhash.1
/usr/info/ispell.info.gz
/usr/doc/ispell-3.1.18-1/README
```

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

  ```
  rpm -qip 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: Today
  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 enter:

  ```
  rpm -qlp 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. As you use the system you will find many more uses for rpm.

12 Miscellania

12.1 Percentages during installation

The `--percent` option to `rpm -i` causes rpm to report "percentages" and other information as the installation progresses. For example:

```
[root@tweety RPMS]# rpm -i --percent m4-1.4-2.i386.rpm
%f m4:1.4:2
%% 63.5290123303439
%% 64.3769822539107
%% 69.7827905166489
%% 71.1077435222219
%% 84.3481645260395
%% 96.8615175680488
%% 98.9218194917149
%% 100
```

This can be (and is) used by other interfaces to `rpm` to give the user feedback during the installation, with a progress meter for instance.

12.2 FTP Installs

Recent 1.x versions of RPM (written in Perl) support installation of packages specified as URL's. For instance:

```
rpm -ivh ftp://ftp.redhat.com/pub/redhat-2.1/i386/RedHat/RPMS/bison-1.24-1.i386.rpm
```

This feature is not yet implemented in the newest 2.x versions of RPM (written in C).

12.3 Querying RPM Packages on an FTP Server

A stripped down version of `rpm` can be built that only supports package queries. This program can be installed on an FTP server such that an anonymous user can safely query packages available on that server. For example, on `ftp.redhat.com`, one can:

```
ncftp>cd /pub/redhat-2.1/i386/RedHat/RPMS
ncftp>site exec rpmq -qip bison-1.24-2.i386.rpm
200-rpmq -qip bison-1.24-2.i386.rpm
200-Name        : bison                        Distribution: RHCL 2.1
200-Version     : 1.24                               Vendor: Red Hat Software
200-Release     : 2                              Build Date:
200-Install date: <package file queried>         Build Host: porky.redhat.com
200-Group       : Development/Tools
200-Size        : 154944
200-Description : GNU parser generator
200  (end of 'rpmq -qip bison-1.24-2.i386.rpm')
```

Not only does this allow users to query a package for its description and file list before actually downloading it, but it opens up possibilities for a program or system that does automatic smart system upgrades via FTP.

12.4 Search Paths

In your `/etc/rpmrc` file you can specify a search path that rpm can follow to find packages for installation. Consider the following lines in an `/etc/rpmrc` file:

```
search: /usr/local/RPMS
search: /mnt/cdrom/RedHat/RPMS
search: ftp://ftp.redhat.com/pub/current/i386/updates/RPMS
search: ftp://ftp.redhat.com/pub/non-free
search: ftp://ftp.redhat.com/pub/contrib/RPMS
```

Now you can *search* for packages:

```
$ rpm --where rpm-
ftp://ftp.redhat.com/pub/current/i386/updates/RPMS/rpm-1.4.5-1.i386.rpm
ftp://ftp.redhat.com/pub/current/i386/updates/RPMS/rpm-1.4.6-1.i386.rpm
```

and install them when you find them:

```
$ rpm -USvh ftp://ftp.redhat.com/pub/current/i386/updates/RPMS/rpm-1.4.6-1.i386.rpm
```

13 Future Directions

We have quite a few plans for expanding RPM facilities. It should understand read-only `/usr` partitions that are mounted throughout a network. Current versions of `rpm` fail when they cannot create files on `/usr` during package installation and upgrading.

Many Linux users currently run many parts of their system from a CD-ROM to conserve disk space. To facilitate this, RPM could install symlinks for the `/usr` parts of a package which point to appropriate files on a CD-ROM.

RPM should have networking facilities designed into it. Either a new networking protocol or extensions to an existing protocol (such as `http`) need to be designed to allow remote querying, installation, and removal of packages. Of course, this raises serious security concerns that need to be addressed.

One of the most popular features of many other packaging systems (including Debian's `dpkg`) is package dependencies. These allow the system to tell a user what parts of the system need to be installed for a particular package to work. The original RPM had no dependency facility. However, the final release of RPM 2.0 will include dependencies, though the exact form they will take is not yet determined.

The POSIX committee has recently finalized a standard for packaging systems (P1387.2). We would like to create user commands that are POSIX compliant for performing package manipulation. However, we will probably not adopt the POSIX method for creating packages as it ignores the problems of creating packages directly from source code.

14 Sample Spec Files for Unzip and Python

The spec file for unzip is very simple.

```
Description: unpacks .zip files such as those made by pkzip under DOS
Name: unzip
Version: 5.12
Release: 1
Exclude: axp
Copyright: distributable
Group: Utilities/Archiving
Source: ftp.uu.net:/pub/archiving/zip/unzip512.tar.Z
Patch: unzip512-fsstnd.patch

%prep
%setup
%patch -p1

%build
make linux

%install
make install
strip /usr/bin/unzip
strip /usr/bin/funzip
strip /usr/bin/unzipsfx

%files
/usr/bin/unzip
/usr/bin/funzip
/usr/bin/unzipsfx
/usr/bin/zipinfo
/usr/man/man1/unzip.1
/usr/man/man1/unzipsfx.1
/usr/man/man1/zipinfo.1
/usr/man/man1/funzip.1
```

The spec file for Python is a little more complicated.

```
Description: Very high level scripting language with X interface
Name: python
```

```
Version: 1.3
Release: 3
Copyright: distributable
Group: Development/Languages
Source: ftp.python.org:/pub/python/src/python1.3.tar.gz
Patch0: python-1.3-misc.patch
Patch1: python-1.3-useshlib.patch

%prep
%setup -n Python-1.3
%patch0 -p1

%ifarch i386
%patch1 -p1 -b .noshlib
%endif

%build
# fix configure botches

echo ': ${LINKFORSHARED='-s'}' > config.cache

%ifarch i386
# note the single '>' below!
echo ': ${LDSHARED='gcc -shared'}' > config.cache
echo ': ${LINKFORSHARED='-rdynamic'}' >> config.cache
%endif

echo ': ${ac_cv_func_lstat='yes'}' >> config.cache

MACHDEP=linux1 ./configure --prefix=/usr --with-readline=/usr/lib

LDFLAGS=-s CFLAGS="$RPM_OPT_FLAGS" make

%install
make install
make libinstall
make sharedinstall

strip /usr/bin/python

%files
/usr/bin/python
/usr/lib/python
```

15 Obtaining the RPM Software

The latest version of RPM is always available at `ftp://ftp.redhat.com/rpm`. It is available in `tar.gz` and RPM format. Also available is the RPM-HOWTO, which contains an introduction to the RPM system and presents detailed explanations of a few spec files. The HOWTO is available in ascii, PostScript, dvi, html, and linuxdoc-sgml formats, and is also browseable at `http://www.redhat.com/redhat/docs/HOWTO/RPM-HOWTO`.

16 Conclusions

The original RPM system has proven to be a stable and versatile base for creating and maintaining a mainstream Linux distribution on multiple platforms. With the release of RPM 2.0, we hope to persuade Linux vendors and developers to adopt the use of RPM for building both distributions and individual packages. We are currently working with the maintainers of multiple Linux distributions to make this a reality.

We would also like to see RPM grow beyond the Linux world. We feel its built in support for multiple architectures and operating systems makes RPM uniquely suited to being a standard packaging system for Internet distributed software. We hope the adoption of the GNU General Public License facilitates widespread adoption of RPM.

17 Biographical Information

Marc Ewing received BS in Computer Science from Carnegie Mellon University in 1992, narrowly escaped from IBM after three months of torture at the hands of his captors, and started Red Hat Software in 1993.

Erik Troan received a BS in Computer Science and a BS in Computer Engineering from North Carolina State University in 1995. After performing unnoticed grunt work at a couple of large corporations, he convinced Red Hat software to pay him to play games. He also maintains the sunsite.unc.edu Linux archive.

18 Copyrights and Trademarks