Maximum RPM

Taking the Red Hat Package Manager to the Limit

Edward C. Bailey

February 17, 1997

Originally published under ISBN: 1-888172-78-9

Revision History

1.1, June 1998 (RHDL added, trademarks, addresses updated)
1.0, February 1997 (Original version)

Red Hat Software, Inc.
4201 Research Commons, Suite 100
79 T. W. Alexander Drive Research Triangle Park, NC 27709
(919) 547-0012
redhat@redhat.com
http://www.redhat.com

To Deb — My lover, editor, indexer, and friend.
To Matt, who heard, "Daddy can't play right now" far too often…

Contents

Preface

0.1 Linux and RPM - A Brief History

Welcome! This is a book about the Red Hat Package Manager or, as it is known to it's friends, RPM. The history of RPM is inextricably linked to the history of Linux, so a bit of Linux history may be in order. Linux is a full-featured implementation of a UNIX-like operating system, and has taken the computing world by storm.

And for good reason — With the addition of Linux, an Intel-based personal computer that had previously been prisoner of the dreaded Windows hourglass is transformed into a fully multitasking, network capable, personal *workstation*. All for the cost of the time required to download, install, and configure the software.

Of course, if you're not the type to tinker with downloaded software, many companies have created CDROMs containing Linux and associated software. The amount of tinkering required with these *distributions* has varied widely. The phrase "You get what you pay for" is never more true than in the area of Linux distributions.

One distribution bears the curious name "Red Hat Linux". Produced by a company of the same name, this Linux distribution was different. One of the key decisions a new Linux user needs to make is which of the many different parts of the distribution to install on their system. Most distributions use some sort of menu, making it easy to pick and choose. Red Hat Linux is no different.

But what *is* different about Red Hat Linux is that the creators of the distribution wanted their customers to have the the ability to make the same choices long *after* the installation process was over. Some commercial UNIX systems have this capability (called "package management"), and a few Linux distributors were trying to come up with something similar, but none had the extensive scope present in RPM.

Over time, Red Hat Linux has become the most popular distribution available today. For it to edge out the previous leader (known as Slackware) in just two years is amazing. There has to be a reason for this kind of success, and a good part of the reason is RPM. But until now, there has been precious little in terms of RPM documentation. You could say that RPM's ease of use has made detailed instructions practically unnecessary, and you'd be right.

However, there are always people that want to know more about their computers, and given the popularity of Red Hat Linux, this alone would have made a book on RPM worthwhile.

But there's more to the story than that.

There is a truism in the world of free software, that goes something like this: If there's a better solution freely available, use it! RPM is no exception to the rule. Put under the terms of the GNU General Public License (Meaning: RPM cannot be made proprietary by *anyone*, not even Bill Gates), RPM started to attract the attention of others in the Linux, Unix, and free software communities.

At present, RPM is used by several commercial software companies producing Linux applications. They find that RPM makes it easier to get their products into the hands of their customers. They also find that it can even make the process of *building* their software easier. (Those of you that develop software for fun and profit, stick around - the second half of this book will show you everything you need to know to get your software "RPM-ized")

People have also ported RPM to several commercial UNIX systems, including DEC's Digital Unix, IBM's AIX, and Silicon Graphics' IRIX. Why? The simple answer is that it makes it easier to install, upgrade, and de-install software. If all these people are using RPM, shouldn't you?

0.2 Parts of the book, and Who They're For

This book is divided into two major sections. The first section is for anyone that needs to know how to use RPM on their system. Given the state of the Linux arena today, this could include just about anyone, including people that are new to Linux, or even UNIX. So those of you that think that

```
ls -FAl !*|less
```

is serious magic (or maybe even a typing error), relax — we'll explain everything you'll need to know in the first section.

In the book's second half, we'll be covering all there is to know about building packages using RPM. Since software engineering on Linux and UNIX systems requires in-depth knowledge of the operating system, available tools, and basic programming concepts, we're going to assume that the reader has sufficient background in these areas. Feel free to browse through the second half, but don't hesitate to seek additional sources of information if you find the going a bit tough.

0.3 Acknowledgements

Writing a book is similar to entering a long-term relationship with an obsessive partner. Throughout the nine months it took to write this book, life went on: job changes, births, deaths, and even a hurricane. Throughout it all, the book demanded constant attention. Therefore, I'd like to thank the people that made it possible to focus on the book to the exclusion of nearly everything else. My wife, Deb and son, Matt supported and encouraged me throughout, even when I was little more than a reclusive houseguest hunched over the computer in the study. Additionally, Deb acted as my editor and indexer, eventually reading the book completely three times! Thank you both.

Thanks also to Marc Ewing and Erik Troan, RPM architects extraordinaire. Without their programming savvy, RPM wouldn't be the elegant tool it is. Without their boundless patience, my many questions would have gone unanswered, and this book would have been much less than it is now. I hope you find this book a worthy companion to your programming handiwork.

Rik Faith provided some much-needed information about PMS and PM, two of RPM's ancestors. Thank you!

Finally a great big thank you goes to Jessica and the gang at L'il Dinos, Jennifer and her crew at the Cary Barnes & Noble coffee shop, and Mom and her "kids" at Schlotzsky's Deli in Durham. If all of you hadn't let me sit around for hours writing, this book wouldn't be nearly as fat as it is. And neither would I!

February, 1997
Cary, North Carolina

Part I

RPM and Computer Users – How to Use RPM to Effectively Manage Your Computer

Chapter 1

An Introduction to Package Management

1.1 What are Packages, and Why Manage Them?

To answer that question, let's go back to the basics for a moment. Computers process information. In order for this to happen, there are some prerequisites:

- A computer (*Obviously!*).
- Some information to process (Also obvious!).
- A program to do the processing (Still pretty obvious!).

Unless these three things come together very little is going to happen, information processing-wise. But each of these items have their own requirements that need to be satisfied before things can get exciting.

Take the computer, for example. While it needs things like electricity and a cool, dry place to operate, it also needs access to the other two items — information and programs — in order to do its thing. The way to get information and programs into a computer is to place them in the computer's mass storage. These days, mass storage invariably means a disk drive. Putting information and programs on the disk drive means that they are stored as files. So much for the computer's part in this.

OK, let's look at the information. Does information have any particular needs? Well, it needs sufficient space on the disk drive, but more importantly, it needs to be in the proper format for the program that will be processing it. That's it for information.

Finally, we have the program. What does it need? Like the information, it needs sufficient disk space on the disk drive. But there are many other things that it may need:

- It may need information to process, in the correct format, named properly, and in the appropriate area on a disk drive somewhere.

- It may need one or more configuration files. These are files that control the program's behavior and permit some level of customization. Like the information, these files must be in the proper format, named properly, and in the appropriate area on a disk. We'll be referring to them by their other name — *config* files — throughout the book.

- It may need work areas on a disk, named properly, and located in the appropriate area.

- It may even need other programs, each with their *own* requirements.

- Although not strictly required by the program itself, the program may come with one or more files containing documentation. These files can be very handy for the humans trying to get the program to do their bidding!

As you can imagine, this can get pretty complicated. It's not so bad once everything is set up properly, but how do things get set up properly in the first place? There are two possibilities:

1. After reading the documentation that comes with the program you'd like to use, you copy the various programs, configuration files, and information onto your computer, making sure they are all named correctly, are located in the proper place, and that there is sufficient disk space to hold them all. You make the appropriate changes to the configuration file(s). Finally, you run any setup programs that are necessary, giving them whatever information they require to do their job.

2. You let the computer do it.

If it seems like the first choice isn't so bad, consider how many files you'll need to keep track of. On a typical Linux system, it's not unusual to have over 20,000 different files. That's a *lot* of documentation reading, file copying, and configuring! And what happens when you want a newer version of a program? More of the same!

Some people think the second alternative is easier. RPM was made for them.

1.1.1 Enter the Package

When you consider that computers are very good at keeping track of large amounts of data, the idea of giving your computer the job of riding herd over 20,000 files seems like a good one. And that's exactly what package management software does. But what *is* a "package"?

A package in the computer sense is very similar to a package in the physical sense. Both are methods of keeping related objects together in the same place. Both need to be opened before the contents can be used. Both can have a "packing slip" taped to the side, identifying the contents.

Normally, package management systems take all the various files containing programs, data, documentation, and configuration information, and place them in one specially formatted file — a package file. In the case of RPM, the package file is sometimes called a "package", a ".rpm file", or even an "RPM". All mean the same thing — a package containing software meant to be installed using RPM.

What types of software are normally found in a package? There are no hard and fast rules, but normally a package's contents consist of one of the following types of software:

- A collection of one or more programs that perform a single well-defined task. This is normally what people think of as an "application". Word processors and programming languages would fit into this category.
- A specific part of an operating system. Examples might be system initialization scripts, a particular command shell, or the software required to support a web server, for example.

Advantages of a Package

One of the most obvious benefits to having a package is that the package is one easily manageable chunk. If you move it from one place to another, there's no risk of any part getting left behind. But although this is the most obvious advantage, it's not the biggest one.

The biggest advantage is that the package can contain the *knowledge* about what it takes to install itself on your computer. And if the package contains the steps required to install itself, the package can also contain the steps required to uninstall itself. What used to be a painful manual process is now a straightforward procedure. What used to be a mass of 20,000 files becomes a couple hundred packages.

1.1.2 Manage Your Packages, or They Will Manage You

A couple *hundred*? Even though the use of packages has decreased the complexity of managing a system by an order of magnitude, it hasn't yet gotten to the level of being a "no-brainer". It's still necessary to keep track of what packages are installed on your system. And if there are some packages that require other packages in order to install or operate correctly, these should be tracked as well.

Packages Lead Active Lives

If you start looking at a computer system as a collection of packages, you'll find that a distinct set of operations will take place on those packages time and time again:

- New packages are installed. Maybe it's a spreadsheet you'll use to keep track of expenses, or the latest shoot-em-up game, but in either case it's new and you want it.
- Old packages are replaced with newer versions. Whoever wrote the word processor you use daily, comes out with a new version. You'll probably want to install the new version and remove the old one.

- Packages are removed entirely. Perhaps that over-hyped strategy game just didn't cut it. You have better things to do with that disk space, so get rid of it!

With this much activity going on, it's easy to lose track of things. What types of package information should be available to keep you informed?

Keeping Track of Packages

Just as there are certain operations that are performed on packages, there are also certain types of information that will make it easier to make sense of the packages installed on your system:

- Certainly you'd like to be able to see what packages are installed. It's easy to forget if that fax program you tried a few months ago is still installed or not.
- It would be nice to be able to get more detailed information on a specific package. This might consist of anything from the date the package was installed, to a list of files it installed on your system.
- Being able to access this information in a variety of ways can be helpful, too. Instead of simply finding out what files a package installed, it might be handy to be able to name a particular file and find out which package installed it.
- If this amount of detail is possible, then it should be possible to see if the way a package is presently installed varies from the way it was originally installed. It's not at all unusual to make a mistake and delete one file — or a hundred. Being able to tell if one or more packages are missing files could greatly simplify the process of getting an ailing system back on its feet again.
- Files containing configuration information can be a real headache. If it were possible to pay extra attention to these files and make sure any changes made to them weren't lost, life would certainly be a lot easier.

1.2 Package Management: How to Do It?

Well, all that sounds great — easy install, upgrade, and deletion of packages; getting package information presented several different ways; making sure packages are installed correctly; and even tracking changes to config files. But how do you do it?

As mentioned above, the obvious answer is to let the computer do it. Many groups have tried to create package management software. There are two basic approaches:

1. Some package management systems concentrate on the specific steps required to manipulate a package.
2. Other package management systems take a different approach, keeping track of the files on the system and manipulating packages by concentrating on the files involved.

Each approach has its good and bad points. In the first method, it's easy to install new packages, somewhat difficult to remove old ones, and almost impossible to obtain any meaningful information about installed packages.

The second method makes it easy to obtain information about installed packages, and fairly easy to install and remove packages. The main problem using this method is that there may not be a well-defined way to execute any commands required during the installation or removal process.

In practice, no package management system uses one approach or the other — all are a mixture of the two. The exact mix and design goals will dictate how well a particular package management system meets the needs of the people using it. At the time Red Hat Software started work on their Linux distribution, there were a number of package management systems in use, each with a different approach to making package management easier.

1.2.1 Ancestors of RPM

Since this is a book on the Red Hat Package Manager, a good way to see what RPM is all about is to look at the package management software that preceded RPM.

RPP

RPP was used in the first Red Hat Linux distributions. Many of RPP's features would be recognizable to anyone who has worked with RPM. Some of these innovative features are:

- Simple, one command installation and uninstallation of packages.
- Scripts that can run before and after installation and uninstallation of packages.
- Package verification. The files of individual packages can be checked to see that they haven't been modified since they were installed.
- Powerful querying. The database of packages can be queried for information about installed packages, including file lists, descriptions and more.

While RPP possessed several of the features that were important enough to continue on as parts of RPM today, it had some weaknesses, too:

- It didn't use "pristine sources". Every program is made up of programming language statements stored in files. This source code is later translated into a binary language that the computer can execute. In the case of RPP, its packages were based on source code that had been modified specifically for RPP, hence the sources weren't pristine. This is a bad idea for a number of fairly technical reasons. Not using pristine sources made it difficult for package developers to keep things straight, particularly if they were building hundreds of different packages.

- It couldn't guarantee executables were built from packaged sources. The process of building a package for RPP was such that there was no way to ensure the executable programs were built from the source code contained in an RPP source package. Once again, this was a problem for the package builder, especially those who had large numbers of packages to build.

- It had no support for multiple architectures. As people started using RPP, it became obvious that the package managers that were unable to simplify the process of building packages for more than one architecture, or type of computer, were going to be at a disadvantage. This was a problem, particularly for Red Hat Software, as they were starting to look at the possibility of creating Linux distributions for other architectures, such as the Digital Alpha.

Even with these problems, RPP was one of the things that made the first Red Hat Linux distributions unique. Its ability to simplify the process of installing software was a real boon to many of Red Hat's customers, particularly those with little experience in Linux.

PMS

While Red Hat Software was busy with RPP, another group of Linux devotees were hard at work with their package management system. Known as PMS, its development, lead by Rik Faith, attacked the problem of package management from a slightly different viewpoint.

Like RPP, PMS was used to package a Linux distribution. This distribution was known as the BOGUS distribution, and all the software in it was built from original unmodified sources. Any changes that were required were patched in during the processing of building the software. This is the concept of "pristine sources" and is PMS's most important contribution to RPM. The importance of pristine sources can not be overstated. It allows the packager to quickly release new version of software, and to immediately see what changes were made to the software.

The chief disadvantages of PMS were weak querying ability, no package verification, no multiple architecture support, and poor database design.

PM

Later, Rik Faith and Doug Hoffman, working under contract for Red Hat Software, produced PM. The design combined all the important features of RPP and PM, including one command installation and uninstallation, scripts run before and after installation and uninstallation, package verification, advanced querying, and pristine sources. However it retained RPP's and PM's chief disadvantages: weak database design and no support for multiple architectures.

PM was very close to a viable package management system, but it wasn't quite ready for prime time. It was never used in a commercially available product.

RPM Version 1

With two major forays into package management behind them, Marc Ewing and Erik Troan went to work on a third attempt. This one would be called the Red Hat Package Manager, or RPM.

Although it built on the experiences of PM, PMS, and RPP, RPM was quite different under the hood. Written in the Perl programming language for fast development, the creation of RPM version 1 focused on addressing the flaws of its ancestors. In some cases, the flaws were eliminated, while in others, the problems remained.

Some of the successes of RPM version 1 were:

- Automatic handling of configuration files. The contents of config files are often changed from what they were in the original package, making it hard for a package manager to know how a particular config file should be handled during installs, upgrades, and erasures. PM made an attempt at config file handling, but in RPM it was improved further. In many respects, this feature is the key to RPM's power and flexibility.
- Ease of rebuilding large numbers of packages. By making it easy for people who were trying to create a Linux distribution consisting of several hundred packages, RPM was a step in the right direction.
- It was easy to use. Many of the concepts used in RPP had withstood the test of time and were used in RPM. For instance, the ability to verify the installation of a package was one of the features that set RPP apart. It was adapted and expanded in RPM version 1.

But RPM version 1 wasn't perfect. There were a number of flaws, some of them major:

- It was slow. While the use of Perl made RPM's development proceed more quickly, it also meant that RPM wouldn't run as quickly as it would have, had it been written in C.
- Its database design was fragile. Unfortunately, under RPM version 1 it was not unusual for there to be problems with the database. While the approach of dedicating a database to package management was a good idea, the implementation used in RPM version 1 left a lot to be desired.
- It was big. This is another artifact of using Perl. Normally, RPM's size requirements were not an issue, except for one area. When performing an initial system install, RPM was run from a small, floppy-based system environment. The need to have Perl available meant space on the boot floppies was always a problem.
- It didn't support multiple architectures (types of computers) well. The need to have a package manager support more than one *type* of computer hadn't been acknowledged before. With RPM version 1, an initial stab was taken at the problem, but the implementation was incomplete. Nonetheless, RPM had been ported to a number of other computer systems. It was becoming obvious that the issue of multi-architecture support was not going away and had to be addressed.

- The package file format wasn't extensible. This made it very difficult to add functionality, since any change to the file format would cause older versions of RPM to break.

Even though their Linux distribution was a success, and RPM was much of the reason for it, Marc and Erik knew that some changes were going to be necessary to carry RPM to the next level.

The RPM of Today: Version 2

Looking back on their experiences with RPM version 1, Marc and Erik made a major change to RPM's design: They rewrote it entirely in C. This did wonderful things to RPM's speed and size. Querying the database was quicker now, and there was no need to have Perl around just to do package management.

In addition, the database format was redesigned to improve both performance and reliability. Displaying package information can take as little as a tenth of the time spent in RPM version 1, for example.

Realizing RPM's potential in the non-Linux arena, they also created rpmlib, a library of RPM routines that allow the use of RPM functionality in other programs. RPM's ability to function on more than one architecture was also enhanced. Finally, the package file format was made more extensible, clearing the way for future enhancements to RPM.

So is RPM perfect? No program can ever reach perfection, and RPM is no exception. But as a package manager that can run on several different types of systems, RPM has a lot to offer, and it will only get better. Let's take a look at the design criteria that drove the development of RPM.

1.3 RPM Design Goals

The design goals of RPM could best be summed up with the phrase "something for everyone". While the main reason for the existence of RPM was to make it easier for Red Hat Software to build the several hundred packages that comprised their Linux distribution, it was not the only reason RPM was created. Let's take a look at the various requirements the Red Hat team used in their design of RPM:

1.3.1 Make it easy to get packages on and off the system

As we've seen earlier in this chapter, the act of installing a package can involve many complex steps. Entrusting these steps to a person who may not have the necessary experience is a strategy for failure. So the goal for RPM was to make it as easy as possible for *anyone* to install packages. The same holds true for removing packages. It is a complex and error-prone operation, and one that RPM should handle for the user.

The other side of this issue is that RPM should give the package builder almost total control in terms of *how* the package is installed. The reason for this is simple: if the package builders do their homework, their package should install and uninstall properly.

1.3.2 Make it easy to verify a package was installed correctly

Because software problems are a fact of life, the ability to verify the proper installation of a package is vital. If done properly, it should be possible to catch a variety of problems, including things such as missing or modified files.

1.3.3 Make it easy for the package builder

While we're dedicating an entire book to package management, in reality it should be a small portion of the package builder's job. Why? They've got better things to do! If they are the people that are actually creating the software to be packaged, *that's* where they should be spending the majority of their time.

Even if the package builder isn't actually writing software, they still have better things to do than worry about building packages. For instance, they may be responsible for building many packages. The less time spent on building an individual package translates to more packages that can be built.

1.3.4 Make it start with the original source code

Delving a bit more into the package builder's world, it was deemed important that RPM start with the original, unmodified source code. Why is this so important?

Using the original sources makes it possible to separate the changes required to build the package from any changes implemented to fix bugs, add new features, or anything else. This is a good thing for package builders, since many of them are *not* the original authors of the programs they package.

This separation makes it easy, months down the road, to know exactly what changes were made in order to get the package to build. This is important when a new version of the packaged software becomes available. Many times it's only necessary to apply the original "package building" changes to the newer software. At worst, the changes provide a starting point to determine what sorts of things might need to be changed in the new version.

1.3.5 Make it work on different computer architectures

One of the tougher things for a package builder to do is to take a program, make it run on more than one type of computer, and distribute packages for each. Because RPM makes it easy to take a program's original source code, add the changes necessary to get it to build, and produce a package for each architecture in one step, it can be pretty handy.

1.4 What's in a package?

With all the magical things we've claimed that package management software in general (and RPM in particular) can do, you'd think there was a tiny computer guru bundled in every package. However, the reality is not that magical. Here's a quick overview of the more important parts of an RPM package[1].

1.4.1 RPM's Package Labels

Every package built for RPM has to have a specific set of information that uniquely identifies it. We call this information a package label. Here are two sample package labels:

- `nls-1.0-1`
- `perl-5.001m-4`

While these labels look like they have very little in common, in fact they all follow RPM's package labelling convention. There are three different components in every package label. Let's look at each one in order:

Component #1: The Software's Name

Every package label begins with the name of the software. The name may be derived from the name of the application packaged, or it may be a name describing a group of related programs bundled together by the package builder. The software names in the packages listed above are: `nls` and `perl`. As you can see, the software name is separated from the rest of the package label by a dash.

Component #2: The Software's Version

Next in the package label is an identifier that describes the version of the software being packaged. If the package builder bundled a number of related programs together, the software version is probably a number of their own choosing. However, if the package consists of one major application, the software version normally comes directly from the application's developer. The actual version specification is quite flexible, as can be seen in the examples above. The versions shown are: `1.0` and `5.001m`. A dash separates the software version from the remainder of the package label.

Component #3: The Package's Release

The package release is the most unambiguous part of a package label. It is a number chosen by the package builder. It reflects the number of times the package has been rebuilt using the same version software. Normally, the rebuilds are due to bugs uncovered after the package

[1] See Appendix A on page 347 for complete details on the contents of a .rpm file.

has been in use for a while. By tradition, the package release starts at `1`. The package releases in the example above are: `1` and `4`.

1.4.2 Labels And Names: Similar, But Distinct

Package labels are used internally by RPM. For example, if you ask RPM to list every installed package, it will respond with a list of package labels. When a package file is created, part of the filename consists of the package label. There is no technical requirement for this, but it does make it easier to keep track of things.

However, a package file may be renamed, and the new filename won't confuse RPM in the least. That's because the package label is contained within the file. For a fairly technical view of the inside of a package file, refer to Appendix A.

1.4.3 Package-wide Information

Some of the information contained in a package is general in nature. This information includes such items as:

- The date and time the package was built.
- A description of the package's contents.
- The total size of all the files installed by the package.
- Information that allows the package to be grouped with similar packages.
- A digital "signature" that can be used to verify the authenticity and integrity of the package.[2]

1.4.4 Per-file Information

Each package also contains information about every file contained in the package. The information includes:

- The name of every file and where it is to be installed.
- Each file's permissions.
- Each file's owner and group specifications.
- The MD5 checksum of each file.[3]
- The file's contents.

[2]For more information on RPM's signature checking capability, refer to section 7.1 on page 93.
[3]We'll discuss MD5 checksums in greater detail in section 6.1 on page 79.

1.5 Let's Get Started

To summarize, a package management system uses the computer to keep track of all the various bits and pieces that comprise an application or an entire operating system. Most package management systems use a specially formatted file to keep everything together in a single, easily manageable entity, or package. Additionally, package management systems tend to provide one or more of the following functions:

- Installing new packages.
- Removing old packages.
- Upgrading from an old package to a new one.
- Obtaining information about installed packages.

RPM has been designed with Red Hat Software's past package management experiences in mind. PM and RPP provided most of these functions with varying degrees of success. Marc Ewing and Erik Troan have worked hard to make RPM better than its predecessors in every way. Now it's time to see how they did, and learn how to use RPM!

Chapter 2

Using RPM to Install Packages

rpm -i (or `--install`) *options* file1.rpm . . . fileN.rpm

Parameters	
`file1.rpm ... fileN.rpm`	One or more RPM package files (URLs OK)

Install—specific Options		Page
`-h` (or `--hash`)	Print hash marks ("#") during install	20
`--test`	Perform installation tests only	21
`--percent`	Print percentages during install	31
`--excludedocs`	Do not install documentation	28
`--includedocs`	Install documentation	29
`--replacepkgs`	Replace a package with a new copy of itself	22
`--replacefiles`	Replace files owned by another package	23
`--force`	Ignore package and file conflicts	28
`--noscripts`	Do not execute pre- and post-install scripts	31
`--prefix` *<path>*	Relocate package to *<path>* if possible	30
`--ignorearch`	Do not verify package architecture	33
`--ignoreos`	Do not verify package operating system	34
`--nodeps`	Do not check dependencies	27
`--ftpproxy` *<host>*	Use *<host>* as the FTP proxy	33
`--ftpport` *<port>*	Use *<port>* as the FTP port	33

General Options		Page
`-v`	Display additional information	19
`-vv`	Display debugging information	21
`--root` *<path>*	Set alternate root to *<path>*	32
`--rcfile` *<rcfile>*	Set alternate rpmrc file to *<rcfile>*	32
`--dbpath` *<path>*	Use *<path>* to find the RPM database	32

2.1 `rpm -i` — What does it do?

Of the many things RPM can do, probably the one that people think of first is the installation of software. As mentioned earlier, installing new software is a complex, error-prone job. RPM turns that process into a single command.

`rpm -i` (`--install` is equivalent) installs software that's been packaged into an RPM package file. It does this by:

- Performing dependency checks.
- Checking for conflicts.
- Performing any tasks required before the install.
- Deciding what to do with config files.
- Unpacking files from the package and putting them in the proper place.
- Performing any tasks required after the install.
- Keeping track of what it did.

Let's go through each of these steps in a bit more detail.

Performing dependency checks: Some packages will not operate properly unless some other package is installed, too. RPM makes sure that the package being installed will have its dependency requirements met. It will also insure that the package's installation will not cause dependency problems for other already-installed packages.

Checking for conflicts: RPM performs a number of checks during this phase. These checks look for things like attempts to install an already installed package, attempts to install an older package over a newer version, or the possibility that a file may be overwritten.

Performing any tasks required before the install: There are cases where one or more commands must be given prior to the actual installation of a package. RPM performs these commands exactly as directed by the package builder, thus eliminating a common source of problems during installations.

Deciding what to do with config files: One of the features that really sets RPM apart from other package managers, is the way it handles configuration files. Since these files are normally changed to customize the behavior of installed software, simply overwriting a config file would tend to make people angry — all their customizations would be gone! Instead, RPM analyzes the situation and attempts to do "the right thing" with config files, even if they weren't originally installed by RPM![1]

[1] Are you interested in what exactly "the right thing" means? Turn to page 44 and read section 4.1.1 for more details.

Unpacking files from the package and putting them in the proper place: This is the step most people think of when they think about installing software. Each package file contains a list of files that are to be installed, as well as their destination on your system. In addition, many other file attributes, such as permissions and ownerships, are set correctly by RPM.

Performing any tasks required after the install: Very often a new package requires that one or more commands be executed after the new files are in place. An example of this would be running `ldconfig` to make new shared libraries accessible.

Keeping track of what it did: Every time RPM installs a package on your system, it keeps track of the files it installed, in its database. The database contains a wealth of information necessary for RPM to do its job. For example, RPM uses the database when it checks for possible conflicts during an install.

2.2 Performing an Install

Let's have RPM install a package. The only thing necessary is to give the command (`rpm -i`) followed by the name of the package file:

```
# rpm -i eject-1.2-2.i386.rpm
#
```

At this point, all the steps outlined above have been performed. The package is now installed. Note that the file name need not adhere to RPM's file naming convention:

```
# mv eject-1.2-2.i386.rpm baz.txt
# rpm -i baz.txt
#
```

In this case, we changed the name of the package file `eject-1.2-2.i386.rpm` to `baz.txt` and then proceeded to install the package. The result is identical to the previous install, that is, the `eject-1.2-2` package successfully installed. The name of the package file, although normally incorporating the package label, is not used by RPM during the installation process. RPM uses the contents of the package file, which means that even if the file was placed on a DOS floppy and the name truncated, the installation would still proceed normally.

2.2.1 URLs — Another Way to Specify Package Files

If you've surfed the World Wide Web, you've no doubt noticed the way web pages are identified:

`http://www.redhat.com/support/docs/rpm/RPM-HOWTO/RPM-HOWTO.html`

This is called a Uniform Resource Locator, or URL. RPM can also use URLs, although they look a little bit different. Here's one:

`ftp://ftp.redhat.com/pub/redhat/code/rpm/rpm-2.3-1.i386.rpm`

The `ftp:` signifies that this URL is a File Transfer Protocol URL. As the name implies, this type of URL is used to move files around. The section containing `ftp.redhat.com` specifies the hostname, or the name of the system where the package file resides.

The remainder of the URL (`/pub/redhat/code/rpm/rpm-2.3-1.i386.rpm`) specifies the path to the package file, followed by the package file itself.

RPM's use of URLs gives us the ability to install a package located on the other side of the world, with a single command:

```
# rpm -i ftp://ftp.gnomovision.com/pub/rpms/foobar-1.0-1.i386.rpm
#
```

This command would use anonymous FTP to obtain the `foobar` version 1.0 package file and install it on your system. Of course, anonymous FTP (no username and password required) is not always available. Therefore, the URL may also contain a username and password preceding the hostname:

`ftp://smith:mypass@ftp.gnomovision.com/pub/rpms/foobar-1.0-1.i386.rpm`

However, entering a password where it can be seen by anyone looking at your screen is a bad idea. So try this format:

`ftp://smith@ftp.gnomovision.com/pub/rpms/foobar-1.0-1.i386.rpm`

RPM will prompt you for your password, and you'll be in business:

```
# rpm -i ftp://smith@ftp.gnomovision.com /pub/rpms/apmd-2.4-1.i386.rpm
Password for smith@ftp.gnomovision.com:  mypass (not echoed)
#
```

After entering a valid password, RPM installs the package.

On some systems, the FTP daemon doesn't run on the standard port 21. Normally this is done for the sake of enhanced security. Fortunately, there is a way to specify a non-standard port in a URL:

`ftp://ftp.gnomovision.com:1024/pub/rpms/foobar-1.0-1.i386.rpm`

This URL will direct the FTP request to port 1024. The `--ftpport` option is another way to specify the port. This option is discussed later in the chapter, on page 33.

2.2.2 A warning message you might never see

Depending on circumstances, the following message might be rare or very common. While performing an ordinary install, RPM prints a warning message:

```
# rpm -i cdp-0.33-100.i386.rpm
warning:  /etc/cdp-config saved as /etc/cdp-config.rpmorig
#
```

What does it mean? It has to do with RPM's handling of config files. In the example above, RPM found a file (`/etc/cdp-config`) that didn't belong to any RPM-installed package. Since the `cdp-0.33-100` package contains a file of the same name that is to be installed in the same directory, there is a problem.

RPM solves this the best way it can. It performs two steps:

1. It renames the original file to `cdp-config.rpmorig`.
2. It installs the new `cdp-config` file that came with the package.

Continuing our example, if we look in `/etc`, we see that this is exactly what has happened:

```
# ls -al /etc/cdp*
-rw-r--r--   1 root     root      119 Jun 23 16:00 /etc/cdp-config
-rw-rw-r--   1 root     root       56 Jun 14 21:44 /etc/cdp-config.rpmorig
#
```

This is the best possible solution to a tricky problem. The package is installed with a config file that is known to work. After all, the original file may be for an older, incompatible version of the software. However, the original file is saved so that it can be studied by the system administrator, who can decide whether the original file should be put back into service or not.

2.3 Two handy options

There are two options to `rpm -i` that work so well, and are so useful, you might think they should be RPM's default behavior. They aren't, but using them only requires that you type an extra two characters:

2.3.1 Getting a bit more feedback with `-v`

Even though `rpm -i` is doing many things, it's not very exciting, is it? When performing installs, RPM is pretty quiet, unless something goes wrong. However, we can ask for a bit more output by adding `-v` to the command:

```
# rpm -iv eject-1.2-2.i386.rpm
Installing eject-1.2-2.i386.rpm
#
```

By adding `-v`, RPM displayed a simple status line. Using `-v` is a good idea, particularly if you're going to use a single command to install more than one package:

```
# rpm -iv *.rpm
Installing eject-1.2-2.i386.rpm
Installing iBCS-1.2-3.i386.rpm
Installing logrotate-1.0-1.i386.rpm
#
```

In this case, there were three .rpm files in the directory. By using a simple wildcard, it's as easy to install one package as it is to install one hundred!

2.3.2 `-h`: Perfect for the Impatient

Sometimes a package can be quite large. Other than watching the disk activity light flash, there's no assurance that RPM is working, and if it is, how far along it is. If you add `-h`, RPM will print fifty hash marks ("`#`") as the install proceeds:

```
# rpm -ih eject-1.2-2.i386.rpm
##################################################
#
```

Once all fifty hash marks are printed, the package is completely installed. Using `-v` with `-h` results in a very nice display, particularly when installing more than one package:

```
# rpm -ivh *.rpm
eject           ##################################################
iBCS            ##################################################
logrotate       ##################################################
#
```

2.4 Additional options to `rpm -i`

Normally `rpm -i`, perhaps with the `-v` and `-h`, is all you'll need. However, there may be times when a basic install is not going to get the job done. Fortunately, RPM has a wealth of install options to make the tough times a little easier. As with any other powerful tool, you should understand these options before putting them to use. Let's take a look at them:

2.4.1 Getting a *lot* more information with `-vv`

Sometimes it's necessary to have even *more* information than we can get with `-v`. By adding another `v`, we can start to see more of RPM's inner workings:

```
# rpm -ivv eject-1.2-2.i386.rpm
D: installing eject-1.2-2.i386.rpm
Installing eject-1.2-2.i386.rpm
D: package:  eject-1.2-2 files test = 0
D: running preinstall script (if any)
D: setting file owners and groups by name (not id)
D: ///usr/bin/eject owned by root (0), group root (0) mode 755
D: ///usr/man/man1/eject.1 owned by root (0), group root (0) mode 644
D: running postinstall script (if any)
#
```

The lines starting with `D:` have been added by using `-vv`. The line ending with "`files test = 0`", means that RPM is actually going to install the package. If the number were non-zero, it would mean that the `--test` option was present, and RPM would not actually perform the installation. For more information on using `--test` with `rpm -i`, see section 2.4.2.

Continuing with the above example, we see that RPM next executes a pre-install script (if there is one), followed by the actual installation of the files in the package. There is one line for each file being installed, and that line shows the filename, ownership, group membership, and permissions (or mode) applied to the file. With larger packages, the output from `-vv` can get quite lengthy! Finally, RPM runs a post-install script, if one exists for the package. We'll be discussing pre- and post-install scripts in more detail in section 2.4.10 on page 31.

In the vast majority of cases, it will not be necessary to use `-vv`. It is normally used by software engineers working on RPM itself, and the output can change without notice. However, it's a handy way to gain insights into RPM's inner workings.

2.4.2 `--test`: Perform Installation Tests Only

There are times when it's more appropriate to take it slow and not try to install a package right away. RPM provides the `--test` option for that. As the names implies, it performs all the checks that RPM normally does during an install, but stops short of actually performing the steps necessary to install the package:

```
# rpm -i --test eject-1.2-2.i386.rpm
#
```

Once again, there's not very much output. This is because the test succeeded; had there been a problem, the output would have been a bit more interesting. In this example, there are some problems:

```
# rpm -i --test rpm-2.0.11-1.i386.rpm
/bin/rpm conflicts with file from rpm-2.3-1
/usr/bin/gendiff conflicts with file from rpm-2.3-1
/usr/bin/rpm2cpio conflicts with file from rpm-2.3-1
/usr/bin/rpmconvert conflicts with file from rpm-2.3-1
/usr/man/man8/rpm.8 conflicts with file from rpm-2.3-1
error:  rpm-2.0.11-1.i386.rpm cannot be installed
#
```

If you'll note the version numbers, we're trying to install an older version of RPM (2.0.11) "on top of" a newer version(2.3). RPM faithfully reported the various file conflicts and summarized with a message saying that the install would not have proceeded, even if `--test` had not been on the command line.

The `--test` option will also catch dependency-related problems:

```
# rpm -i --test blather-7.9-1.i386.rpm
failed dependencies:
bother >= 3.1 is needed by blather-7.9-1
#
```

Here's a tip for all you script-writers out there: RPM will return a non-zero status if the `--test` option detects problems…

2.4.3 `--replacepkgs`: Install the Package Even If Already Installed

The `--replacepkgs` option is used to force RPM to install a package that it believes to be installed already. This option is normally used if the installed package has been damaged somehow and needs to be fixed up.

To see how the `--replacepkgs` option works, let's first install some software:

```
# rpm -iv cdp-0.33-2.i386.rpm
Installing cdp-0.33-2.i386.rpm
#
```

OK, now that we have `cdp-0.33-2` installed, let's see what happens if we try to install the same version "on top of" itself:

```
# rpm -iv cdp-0.33-2.i386.rpm
Installing cdp-0.33-2.i386.rpm
```

```
package cdp-0.33-2 is already installed
error:  cdp-0.33-2.i386.rpm cannot be installed
#
```

That didn't go very well. Let's see what adding `--replacepkgs` will do :

```
# rpm -iv --replacepkgs cdp-0.33-2.i386.rpm
Installing cdp-0.33-2.i386.rpm
#
```

Much better. The original package was replaced by a new copy of itself.

2.4.4 `--replacefiles`: Install the Package Even If It Replaces Another Package's Files

While the `--replacepkgs` option permitted a package to be installed "on top of" itself, `--replacefiles` is used to allow a package to overwrite files belonging to a different package. Sounds strange? Let's go over it in a bit more detail.

One thing that sets RPM apart from many other package managers is that it keeps track of all the files it installs in a database. Each file's database entry contains a variety of information about the file, including a means of summarizing the file's contents.[2] By using these summaries, known as *MD5 checksums*, RPM can determine if a particular file is going to be replaced by a file with the same name, but different contents. Here's an example:

Package "A" installs a file (we'll call it `/bin/foo.bar`). Once Package A is installed, `foo.bar` resides happily in the `/bin` directory. In the RPM database, there is an entry for `/bin/foo.bar`, including the file's MD5 checksum.

However, there is a another package, "B". Package B also has a file called `foo.bar` that *it* wants to install in `/bin`. There can't be two files in the same directory with the same name. The files are different; their MD5 checksums do not match. What happens if Package B is installed? Let's find out. Here, we've installed a package:

```
# rpm -iv cdp-0.33-2.i386.rpm
Installing cdp-0.33-2.i386.rpm
#
```

OK, no problem there. But we have another package to install. In this case, it is a new release of the `cdp` package. It should be noted that RPM's detection of file conflicts does not depend on the two packages being related. It is strictly based on the name of the file, the directory in which it resides, and the file's MD5 checksum. Here's what happens when we try to install the package:

[2] We'll get more into this aspect of RPM in chapter 6 when we discuss `rpm -V`, specifically section 6.1 on page 79

```
# rpm -iv cdp-0.33-3.i386.rpm
Installing cdp-0.33-3.i386.rpm
/usr/bin/cdp conflicts with file from cdp-0.33-2
error:  cdp-0.33-3.i386.rpm cannot be installed
#
```

What's happening? The package `cdp-0.33-2` has a file, `/usr/bin/cdp`, that it installed. Sure enough, there it is. Let's circle the size and creation date of the file for future reference:

```
# ls -al /usr/bin/cdp
-rwxr-xr-x 1 root root 34460 Feb 25 14:27 /usr/bin/cdp
#
```

The package we just tried to install, `cdp-0.33-`*3* (note the different release), also installs a file `cdp` in `/usr/bin`. Since there is a conflict, that means that the two package's `cdp` files must be different — their checksums don't match. Because of this, RPM won't let the second package install. But with `--replacefiles`, we can force RPM to let the `/usr/bin/cdp` from `cdp-0.33-3` replace the `/usr/bin/cdp` from `cdp-0.33-2`:

```
# rpm -iv --replacefiles cdp-0.33-3.i386.rpm
Installing cdp-0.33-3.i386.rpm
#
```

Taking a closer look at `/usr/bin/cdp`, we find that they certainly *are* different, both in size and creation date:

```
# ls -al /usr/bin/cdp
-rwxr-xr-x 1 root root 34444 Apr 24 22:37 /usr/bin/cdp
#
```

File conflicts should be a relatively rare occurrence. They only happen when two packages attempt to install files with the same name but different contents. There are two possible reasons for this to happen:

- Installing a newer version of a package without erasing the older version. A newer version of a package is a *wonderful* source of file conflicts against older versions — the filenames remain the same, but the contents change. We used it in our example because it's an easy way to show what happens when there are file conflicts. However, it is usually a *bad* idea when it comes to doing this as a way to upgrade packages. RPM has a special option for this (`rpm -U`) that is discussed in chapter 4 on page 43.
- Installing two unrelated packages that each install a file with the same name. This may happen because of poor package design (hence the file residing in more than one package), or a lack of coordination between the people building the packages.

`--replacefiles` and Config Files

What happens if a conflicting file is a config file that you've sweated over and worked on until it's just right? Will issuing a `--replacefiles` on a package with a conflicting config file blow all your changes away?

No! RPM won't cook your goose.[3] It will save any changes you've made, to a config file called *file*`.rpmsave`. Let's give it a try:

As system administrator, you want to make sure your new users have a rich environment the first time they log in. So you've come up with a really nifty `.bashrc` file that will be executed whenever they log in. Knowing that *everyone* will enjoy your wonderful `.bashrc` file, you place it in `/etc/skel`. That way, every time a new account is created, your `.bashrc` will be copied into the new user's login directory.

Not realizing that the `.bashrc` file you modified in `/etc/skel` is listed as a config file in a package called (strangely enough) `etcskel`, you decide to experiment with RPM using the `etcskel` package. First you try to install it:

```
# rpm -iv etcskel-1.0-100.i386.rpm
etcskel /etc/skel/.bashrc conflicts with file from etcskel-1.0-3
error:  etcskel-1.0-100.i386.rpm cannot be installed
#
```

Hmmm. That didn't work. Wait a minute! I can add `--replacefiles` to the command and it should install just fine:

```
# rpm -iv --replacefiles etcskel-1.0-100.i386.rpm
Installing etcskel-1.0-100.i386.rpm
warning:  /etc/skel/.bashrc saved as /etc/skel/.bashrc.rpmsave
#
```

Wait a minute... That's my customized `.bashrc`! Was it *really* saved?

```
# ls -al /etc/skel/
total 8
-rwxr-xr-x 1 root root 186 Oct 12 1994 .Xclients
-rw-r--r-- 1 root root 1126 Aug 23 1995 .Xdefaults
-rw-r--r-- 1 root root 24 Jul 13 1994 .bash_logout
-rw-r--r-- 1 root root 220 Aug 23 1995 .bash_profile
-rw-r--r-- 1 root root 169 Jun 17 20:02 .bashrc
-rw-r--r-- 1 root root 159 Jun 17 20:46 .bashrc.rpmsave
drwxr-xr-x 2 root root 1024 May 13 13:18 .xfm
lrwxrwxrwx 1 root root 9 Jun 17 20:46 .xsession -> .Xclients
# cat /etc/skel/.bashrc.rpmsave
```

[3] You'll have to do that yourself!

```
# .bashrc
# User specific aliases and functions
# Modified by the sysadmin
uptime
# Source global definitions
if [ -f /etc/bashrc ]; then
.  /etc/bashrc
fi
#
```

Whew! You heave a sigh of relief, and study the new `.bashrc` to see if the changes need to be integrated into your customized version.

`--replacefiles` Can Mean Trouble Down the Road

While `--replacefiles` can make today's difficult install go away, it can mean big headaches in the future. When the time comes for erasing the packages involved in a file conflict, bad things can happen.

What bad things? Well, files can be deleted. Here's how, in three easy steps:

1. Two packages are installed. When the second package is installed, there is a conflict with a file installed by the first package. Therefore, the `--replacefiles` option is used to force RPM to replace the conflicting file with the one from the second package.
2. At some point in the future, the second package is erased.
3. The conflicting file is gone!

Let's look at an example. First, we install a new package. Next, we take a look at a file it installed, noting the size and creation date.

```
# rpm -iv cdp-0.33-2.i386.rpm
Installing cdp-0.33-2.i386.rpm
# ls -al /usr/bin/cdp
-rwxr-xr-x 1 root root 34460 Feb 25 14:27 /usr/bin/cdp
#
```

Next, we try to install a newer release of the same package. It fails:

```
# rpm -iv cdp-0.33-3.i386.rpm
Installing cdp-0.33-3.i386.rpm
/usr/bin/cdp conflicts with file from cdp-0.33-2
error:  cdp-0.33-3.i386.rpm cannot be installed
#
```

So, we use `--replacefiles` to convince the newer package to install. We note that the newer package installed a file on top of the file originally installed:

```
# rpm -iv --replacefiles cdp-0.33-3.i386.rpm
Installing cdp-0.33-3.i386.rpm
# ls -al /usr/bin/cdp
-rwxr-xr-x 1 root root 34444 Apr 24 22:37 /usr/bin/cdp
#
```

The original `cdp` file, 34,460 bytes long, and dated February 25th, has been replaced with a file with the same name, but 34,444 bytes long from the 24th of April. The original file is long gone.

Next, we erased the package we just installed.[4] Finally, we tried to find the file:

```
# rpm -e cdp-0.33-3
# ls -al /usr/bin/cdp
ls:  /usr/bin/cdp:  No such file or directory
#
```

The file is gone. Why is this? The reason is that `/usr/bin/cdp` from the first package was replaced when the second package was installed using the `--replacefiles` option. Then, when the second package was erased, the `/usr/bin/cdp` file was removed, since it belonged to the second package. If the first package had been erased first, there would have been no problem, since RPM would have realized that the first package's file had already been deleted, and would have left the file in place.

The only problem with this state of affairs is that the first package is still installed, *except* for `/usr/bin/cdp`. So now there's a partially installed package on the system. What to do? Perhaps it's time to exercise your new-found knowledge by issuing an `rpm -i --replacepkgs` command to fix up the first package...

2.4.5 `--nodeps`: Do Not Check Dependencies Before Installing Package

One day it'll happen. You'll be installing a new package, when suddenly, the install bombs:

```
# rpm -i blather-7.9-1.i386.rpm
failed dependencies:
bother >= 3.1 is needed by blather-7.9-1
#
```

What happened? The problem is that the package you're installing requires another package

[4]For more information on erasing packages with `rpm -e`, see chapter 3 on page 35.

to be installed in order for it to work properly. In our example, the `blather` package won't work properly unless the `bother` package (and more specifically, `bother` version 3.1 or later) is installed. Since our system doesn't have an appropriate version of `bother` installed at all, RPM aborted the installation of `blather`.

Now, 99 times out of 100, this exactly the right thing for RPM to do. After all, if the package doesn't have everything it needs to work properly, why try to install it? Well, as with everything else in life, there are exceptions to the rule. And that is why there is a `--nodeps` option.

Adding the `--nodeps` options to an install command directs RPM to ignore any dependency-related problems and to complete the package installation. Going back to our example above, let's add the `--nodeps` option to the command line and see what happens:

```
# rpm -i --nodeps blather-7.9-1.i386.rpm
#
```

The package was installed without a peep. Whether it will work properly is another matter, but it is installed. In general, it's not a good idea to use `--nodeps` to get around dependency problems. The package builders included the dependency requirements for a reason, and it's best not to second-guess them.

2.4.6 `--force`: The Big Hammer

Adding `--force` to an install command is a way of saying "Install it anyway!" In essence, it adds `--replacepkgs` and `--replacefiles` to the command. Like a big hammer, `--force` is an irresistible force[5] that makes things happen. In fact, the only thing that will prevent a `--force`'ed install from proceeding is a dependency conflict.

And like a big hammer, it pays to fully understand why you need to use `--force` before actually using it.

2.4.7 `--excludedocs`: Do Not Install Documentation For This Package

RPM has a number of good features. One of them is the fact that RPM classifies the files it installs into one of three categories:

1. Config files.
2. Files containing documentation.
3. All other files.

RPM uses the `--excludedocs` option to prevent files classified as documentation from being installed. In the following example, we know that the package contains documentation:

[5] No pun intended.

specifically, the man page, /usr/man/man1/cdp.1. Let's see how --excludedocs keeps it from being installed:

```
# rpm -iv --excludedocs cdp-0.33-3.i386.rpm
Installing cdp-0.33-3.i386.rpm
# ls -al /usr/man/man1/cdp.1
ls:  /usr/man/man1/cdp.1:  No such file or directory
#
```

The primary reason to use --excludedocs is to save on disk space. The savings can be sizeable. For example, on an RPM-installed Linux system, there can be over 5,000 documentation files, using nearly 50 megabytes.

If you like, you can make --excludedocs the default for *all* installs. To do this, simply add the following line to /etc/rpmrc, .rpmrc in your login directory, or the file specified with the --rcfile (which is discussed on page 32) option:

```
excludedocs:  1
```

After that, every time an rpm -i command is run, it will not install any documentation files.[6]

2.4.8 --includedocs: Install Documentation For This Package

As the name implies, --includedocs directs RPM to install any files marked as being documentation. This option is normally not required, unless the rpmrc file entry "excludedocs: 1" is included in the referenced rpmrc file. Here's an example. Note that in this example, /etc/rpmrc contains "excludedocs: 1", which directs RPM not to install documentation files:

```
# ls /usr/man/man1/cdp.1
ls:  /usr/man/man1/cdp.1:  No such file or directory
# rpm -iv cdp-0.33-3.i386.rpm
Installing cdp-0.33-3.i386.rpm
# ls /usr/man/man1/cdp.1
ls:  /usr/man/man1/cdp.1:  No such file or directory
#
```

Here we've checked to make sure that the cdp man page did not previously exist on the system. Then after installing the cdp package, we find that the "excludedocs: 1" in /etc/rpmrc did its job: the man page wasn't installed. Let's try it again, this time adding the --includedocs option:

```
# ls /usr/man/man1/cdp.1
ls:  /usr/man/man1/cdp.1:  No such file or directory
```

[6] For more information on rpmrc files, refer to Appendix B on page 363.

```
# rpm -iv --includedocs cdp-0.33-3.i386.rpm
Installing cdp-0.33-3.i386.rpm
# ls /usr/man/man1/cdp.1
-rw-r--r-- 1 root root 4550 Apr 24 22:37 /usr/man/man1/cdp.1
#
```

The `--includedocs` option overrode the rpmrc file's "`excludedocs: 1`" entry, causing RPM to install the documentation file.

2.4.9 `--prefix` *<path>*: Relocate the package to *<path>*, if possible

Some packages give the person installing them flexibility in determining where on their system they should be installed. These are known as relocatable packages. A relocatable package differs from a package that cannot be relocated, in only one way — the definition of a default prefix. Because of this, it takes a bit of additional effort to determine if a package is relocatable. But here's an RPM command that can be used to find out:[7]

```
rpm -qp --queryformat "%{defaultprefix\}\n" <packagefile>
```

Just replace *<packagefile>* with the name of the package file you want to check out. If the package is not relocatable, you'll only see the word `(none)`. If, on the other hand, the command displays a path, that means the package is relocatable. Unless specified otherwise, every file in the package will be installed somewhere below the path specified by the default prefix.

What if you want to specify otherwise? Easy: just use the `--prefix` option. Let's give it a try:

```
# rpm -qp --queryformat "%{defaultprefix}\n" cdplayer-1.0-1.i386.rpm
/usr/local
# rpm -i --prefix /tmp/test cdplayer-1.0-1.i386.rpm
#
```

Here we've used our magic query command to determine that the `cdplayer` package is relocatable. It normally installs below `/usr/local`, but we wanted to move it around. By adding the `--prefix` option, we were able to make the package install in `/tmp/test`. If we take a look there, we'll see that RPM created all the necessary directories to hold the package's files:

```
# ls -lR /tmp/test/
total 2
drwxr-xr-x 2 root root 1024 Dec 16 13:21 bin/
drwxr-xr-x 3 root root 1024 Dec 16 13:21 man/
```

[7] We discuss RPM's query commands in chapter 5.

```
/tmp/test/bin:
total 41
-rwxr-xr-x 1 root root 40739 Oct 14 20:25 cdp*
lrwxrwxrwx 1 root root 17 Dec 16 13:21 cdplay -> /tmp/test/bin/cdp*

/tmp/test/man:
total 1
drwxr-xr-x 2 root root 1024 Dec 16 13:21 man1/

/tmp/test/man/man1:
total 5
-rwxr-xr-x 1 root root 4550 Oct 14 20:25 cdp.1*
#
```

2.4.10 `--noscripts`: Do Not Execute Pre- and Post-install Scripts

Before we talk about the `--noscripts` option, we need to cover a bit of background. In section 2.4.1 on page 21, we saw some output from an install using the `-vv` option. As can be seen, there are two lines that mention pre-install and post-install scripts. When some packages are installed, they may require that certain programs be executed before, after, or before *and* after the package's files are copied to disk.[8]

The `--noscripts` option prevents these scripts from being executed during an install. *This is a very dangerous thing to do!* The `--noscripts` option is really meant for package builders to use during the development of their packages. By preventing the pre- and post-install scripts from running, a package builder can keep a buggy package from bringing down their development system. Once the bugs are found and eliminated, the `--noscripts` option is no longer necessary.

2.4.11 `--percent`: Not Meant for Human Consumption

An option that will probably *never* be very popular is `--percent`. This option is meant to be used by programs that interact with the user, perhaps presenting a graphical user interface for RPM. When the `--percent` option is used, RPM displays a series of numbers. Each number is a percentage that indicates how far along the install is. When the number reaches 100%, the installation is complete.

```
# rpm -i --percent iBCS-1.2-3.i386.rpm
%f iBCS:1.2:3
%% 0.002140
%% 1.492386
%% 5.296632
```

[8]It's possible to use RPM's query command to see if a package has pre- or post-install scripts. See section 5.2.2 on page 66 for more information.

```
%% 9.310026
%% 15.271010
%% 26.217846
%% 31.216000
%% 100.000000
%% 100.000000
#
```

The list of percentages will vary depending on the number of files in the package, but every package ends at 100% when completely installed.

2.4.12 `--rcfile` *<rcfile>*: Use *<rcfile>* As An Alternate `rpmrc` File

The `--rcfile` option is used to specify a file containing default settings for RPM. Normally, this option is not needed. By default, RPM uses `/etc/rpmrc` and a file named `.rpmrc` located in your login directory.

This option would be used if there was a need to switch between several sets of RPM defaults. Software developers and package builders will normally be the only people using the `--rcfile` option. For more information on `rpmrc` files, see Appendix B.

2.4.13 `--root` *<path>*: Use *<path>* As An Alternate Root

Adding `--root` *<path>* to an install command forces RPM to assume that the directory specified by *<path>* is actually the "root" directory. The `--root` option affects every aspect of the install process, so pre- and post-install scripts are run with *<path>* as their root directory (using `chroot(2)`, if you must know). In addition, RPM expects its database to reside in the directory specified by the `dbpath` `rpmrc` file entry, relative to *<path>*.[9]

Normally this option is only used during an initial system install, or when a system has been booted off a "rescue disk" and some packages need to be re-installed.

2.4.14 `--dbpath` *<path>*: Use *<path>* To Find RPM Database

In order for RPM to do its handiwork, it needs access to an RPM database. Normally, this database exists in the directory specified by the `rpmrc` file entry, `dbpath`. By default, `dbpath` is set to `/var/lib/rpm`.

Although the `dbpath` entry can be modified in the appropriate `rpmrc` file, the `--dbpath` option is probably a better choice when the database path needs to be changed temporarily. An example of a time the `--dbpath` option would come in handy is when it's necessary to examine an RPM database copied from another system. Granted, it's not a common occurrence, but it's difficult to handle any other way.

[9] For more information on `rpmrc` file entries, see Appendix B.

2.4.15 `--ftpport` *<port>*: Use *<port>* In FTP-based Installs

Back in section 2.2.1 we showed how RPM can access package files by the use of a URL. We also mentioned that some systems may not use the standard FTP port. In those cases, it's necessary to give RPM the proper port number to use. As we mentioned above, one approach is to embed the port number in the URL itself.

Another approach is to use the `--ftpport` option. RPM will access the desired port when this option, along with the port number, is added to the command line. In cases where the desired port seldom changes, it may be entered in an `rpmrc` file by using the `ftpport` entry.[10]

2.4.16 `--ftpproxy` *<host>*: Use *<host>* As Proxy In FTP-based Installs

Many companies and Internet Service Providers (ISPs) employ various methods to protect their network connections against misuse. One of these methods is to use a system that will process all FTP requests on behalf of the other systems on the company or ISP network. By having a single computer act as a proxy for the other systems, it serves to protect the other systems against any FTP-related misuse.

When RPM is employed on a network with an FTP proxy system, it will be necessary for RPM to direct all its FTP requests to the FTP proxy. RPM will send its FTP requests to the specified proxy system when the `--ftpproxy` option, along with the proxy hostname, is added to the command line.

In cases where the proxy host seldom changes, it may be entered in an `rpmrc` file by using the `ftpproxy` entry.[11]

2.4.17 `--ignorearch`: Do Not Verify Package Architecture

When a package file is created, RPM specifies the architecture, or type of computer hardware, for which the package was created. This is a good thing, as the architecture is one of the main factors in determining whether a package written for one computer is going to be compatible with another computer.

When a package is installed, RPM uses the `arch_compat` `rpmrc` entries in order to determine what are normally considered compatible architectures. Unless you're porting RPM to a new architecture, you shouldn't make any changes to these entries.[12] While RPM attempts to make the right decisions regarding package compatibility, there are times when it errs on the side of conservatism. In those cases, it's necessary to override RPM's decision. The `--ignorearch` option is used in those cases. When added to the command line, RPM will not perform any architecture-related checking.

Unless you really know what you're doing, you should *never* use `--ignorearch`!

[10] The use of `rpmrc` files is described in Appendix B

[11] The use of `rpmrc` files is described in Appendix B

[12] If you *are* porting RPM, you'll find more on `arch_compat` in section 19.3.1 on page 273.

2.4.18 `--ignoreos`: Do Not Verify Package Operating System

When a package file is created, RPM specifies the operating system for which the package was created. This is a good thing as the operating system is one of the main factors in determining whether a package written for one computer is going to be compatible with another computer.

When a package is installed, RPM uses the `os_compat` `rpmrc` entries to determine what are normally considered compatible operating systems. Unless you're porting RPM to a new operating system, you shouldn't make any changes to these entries.[13] While RPM attempts to make the right decisions regarding package compatibility, there are times when it errs on the side of conservatism. In those cases, it's necessary to override RPM's decision. The `--ignoreos` option is used in those cases. When added to the command line, RPM will not perform any operating system-related checking.

Unless you really know what you're doing, you should *never* use `--ignoreos`!

[13]If you *are* porting RPM, you'll find more on `os_compat` in section 19.3.1 on page 273.

Chapter 3

Using RPM to Erase Packages

`rpm -e` (or `--erase`) *options* pkg1 … pkgN

Parameters		
`pkg1 ... pkgN`	One or more installed packages	
Erase—specific Options		Page
`--test`	Perform erase tests only	37
`--noscripts`	Do not execute pre- and post-uninstall scripts	39
`--nodeps`	Do not check dependencies	38
General Options		Page
`-vv`	Display debugging information	36
`--root` *<path>*	Set alternate root to *<path>*	40
`--rcfile` *<rcfile>*	Set alternate rpmrc file to *<rcfile>*	39
`--dbpath` *<path>*	Use *<path>* to find the RPM database	40

3.1 `rpm -e` — What Does it Do?

The `rpm -e` command (`--erase` is equivalent) removes, or erases, one or more packages from the system. RPM performs a series of steps whenever it erases a package:

- It checks the RPM database to make sure that no other packages depend on the package being erased.
- It executes a pre-uninstall script (if one exists).
- It checks to see if any of the package's config files have been modified. If so, it saves copies of them.
- It reviews the RPM database to find every file listed as being part of the package, and if they do not belong to another package, deletes them.
- It executes a post-uninstall script (if one exists).

- It removes all traces of the package (and the files belonging to it) from the RPM database.

That's quite a bit of activity for a single command. No wonder RPM can be such a time-saver!

3.2 Erasing a Package

The most basic erase command is:

```
# rpm -e eject
#
```

In this case, the `eject` package was erased. There isn't much in the way of feedback, is there? Could we get more if we add -v?

```
# rpm -ev eject
#
```

Still nothing. However, there's another option that can be counted on to give a wealth of information. Let's give it a try:

3.2.1 Getting More Information With -vv

By adding `-vv` to the command line, we can often get a better feel for what's going on inside RPM. The `-vv` option was really meant for the RPM developers, and its output may change, but it is a great way to gain insight into RPM's inner workings. Let's try it with `rpm -e`:

```
# rpm -evv eject
D: uninstalling record number 286040
D: running preuninstall script (if any)
D: removing files test = 0
D: /usr/man/man1/eject.1 - removing
D: /usr/bin/eject - removing
D: running postuninstall script (if any)
D: removing database entry
D: removing name index
D: removing group index
D: removing file index for /usr/bin/eject
D: removing file index for /usr/man/man1/eject.1
#
```

Although -v had no effect on RPM's output, -vv gave us a torrent of output. But what does it tell us?

First, RPM displays the package's record number. The number is normally of use only to people that work on RPM's database code.

Next, RPM executes a "pre-uninstall" script, if one exists. This script can execute any commands required to remove the package before any files are actually deleted.

The "files test = 0" line indicates that RPM is to actually erase the package. If the number had been non-zero, RPM would only be performing a test of the package erasure. This happens when the --test option is used. Refer to section 3.3.1 on page 37 for more information on the use of the --test option with rpm -e.

The next two lines log the actual removal of the files comprising the package. Packages with many files can result in a lot of output when using -vv!

Next, RPM executes a "post-uninstall" script, if one exists. Like the pre-uninstall script, this script is used to perform any processing required to cleanly erase the package. Unlike the pre-uninstall script, however, the post-uninstall script runs *after* all the package's files have been removed.

Finally, the last five lines show the process RPM uses to remove every trace of the package from its database. From the messages, we can see that the database contains some per-package data, followed by information on every file installed by the package.

3.3 Additional Options

If you're interested in a complex command with lots of options, rpm -e is *not* the place to look. There just aren't that many different ways to erase a package! But there are a few options you should know about.

3.3.1 --test — Go Through the Process of Erasing the Package, But Do Not Erase It

If you're a bit gun-shy about erasing a package, you can use the --test option first to see what rpm -e would do:

```
# rpm -e --test bother
removing these packages would break dependencies:
bother >= 3.1 is needed by blather-7.9-1
#
```

It's pretty easy to see that the blather package wouldn't work very well if bother were erased. To be fair, however, RPM wouldn't have erased the package in this example unless we used the --nodeps option, which we'll discuss shortly.

However, if there are no problems erasing the package, you won't see very much:

```
# rpm -e --test eject
#
```

We know, based on previous experience, that `-v` doesn't give us any additional output with `rpm -e`. However, we *do* know that `-vv` works wonders. Let's see what *it* has to say:

```
# rpm -evv --test eject
D: uninstalling record number 286040
D: running preuninstall script (if any)
D: would remove files test = 1
D: /usr/man/man1/eject.1 - would remove
D: /usr/bin/eject - would remove
D: running postuninstall script (if any)
D: would remove database entry
#
```

As you can see, the output is similar to that of a regular erase command using the `-vv` option, with the following exceptions:

- The "`would remove files test = 1`" line ends with a non-zero number. This is because `--test` has been added. If the command hadn't included `--test`, the number would have been 0, and the package would have been erased.
- There is a line for each file that RPM would have removed, each one ending with "`would remove`" instead of "`removing`".
- There is only one line at the end, stating: "`would remove database entry`", versus the multi-line output showing the cleanup of the RPM database during an actual erase.

By using `--test` in conjunction with `-vv`, it's easy to see exactly what RPM would do during an actual erase.

3.3.2 `--nodeps`: Do Not Check Dependencies Before Erasing Package

It's likely that one day while erasing a package, you'll see something like this:

```
# rpm -e bother
removing these packages would break dependencies:
bother >= 3.1 is needed by blather-7.9-1
#
```

What happened? The problem is that one or more of the packages installed on your system require the package you're trying to erase. Without it, they won't work properly. In our example, the `blather` package won't work properly unless the `bother` package (and more specifically, `bother` version 3.1 or later) is installed. Since we're trying to erase `bother`, RPM aborted the erasure.

Now, 99 times out of 100, this is exactly the right thing for RPM to do. After all, if the package is needed by other packages, why try to erase it? As with everything else in life, there are exceptions to the rule. And that is why there is a `--nodeps` option.

Adding the `--nodeps` options to an erase command directs RPM to ignore any dependency-related problems, and to erase the package. Going back to our example above, let's add the `--nodeps` option to the command line and see what happens:

```
# rpm -e --nodeps bother
#
```

The package was erased without a peep. Whether the `blather` package will work properly is another matter. In general, it's not a good idea to use `--nodeps` to get around dependency problems. The package builders included the dependency requirements for a reason, and it's best not to second-guess them.

3.3.3 `--noscripts` — Do *Not* Execute Pre- and Post-uninstall Scripts

In section 3.2.1 on page 36, we used the `-vv` option to see what RPM was actually doing when it erased a package. We noted that there were two scripts, a pre-uninstall and a post-uninstall, that were used to execute commands required during the process of erasing a package.

The `--noscripts` option prevents these scripts from being executed during an erase. *This is a very dangerous thing to do!* The `--noscripts` option is really meant for package builders to use during the development of their packages. By preventing the pre- and post-uninstall scripts from running, a package builder can keep a buggy package from bringing down their development system. Once the bugs are found and eliminated, there's very little need to prevent these scripts from running; in fact, doing so can *cause* problems!

3.3.4 `--rcfile` *<rcfile>* — Read *<rcfile>* For RPM Defaults

The `--rcfile` option is used to specify a file containing default settings for RPM. Normally, this option is not needed. By default, RPM uses `/etc/rpmrc` and a file named `.rpmrc` located in your login directory.

This option would be used if there was a need to switch between several sets of RPM defaults. Software developers and package builders will normally be the only people using the `--rcfile` option. For more information on `rpmrc` files, see Appendix B on page 363.

3.3.5 `--root <path>` — Use *<path>* As the Root

Adding `--root <path>` to an install command forces RPM to assume that the directory specified by *<path>* is actually the "root" directory. The `--root` option affects every aspect of the install process, so pre- and post-install scripts are run with *<path>* as their root directory (using `chroot(2)`, if you must know). In addition, RPM expects its database to reside in the directory specified by the `dbpath` `rpmrc` file entry, relative to *<path>*.[1]

Normally this option is only used during an initial system install, or when a system has been booted off a "rescue disk" and some packages need to be re-installed.

3.3.6 `--dbpath <path>`: Use *<path>* To Find RPM Database

In order for RPM to do its handiwork, it needs access to an RPM database. Normally, this database exists in the directory specified by the `rpmrc` file entry, `dbpath`. By default, `dbpath` is set to `/var/lib/rpm`.

Although the `dbpath` entry can be modified in the appropriate `rpmrc` file, the `--dbpath` option is probably a better choice when the database path needs to be changed temporarily. An example of a time the `--dbpath` option would come in handy is when it's necessary to examine an RPM database copied from another system. Granted, it's not a common occurrence, but it's difficult to handle any other way.

3.4 `rpm -e` and Config files

If you've made changes to a configuration file that was originally installed by RPM, your changes won't be lost if you erase the package. Say, for example, that we've made changes to `/etc/skel/.bashrc` (a config file), which was installed as part of the `etcskel` package. Later, we remove `etcskel`:

```
# rpm -e etcskel
#
```

But if we take a look in `/etc/skel`, look what's there:

```
# ls -al
total 5
drwxr-xr-x 3 root root 1024 Jun 17 22:01 .
drwxr-xr-x 8 root root 2048 Jun 17 19:01 ..
-rw-r--r-- 1 root root 152 Jun 17 21:54 .bashrc.rpmsave
drwxr-xr-x 2 root root 1024 May 13 13:18 .xfm
#
```

[1] For more information on `rpmrc` file entries, see Appendix B.

Sure enough: `.bashrc.rpmsave` is a copy of your modified `.bashrc` file! Remember, however, that this feature only works with config files. Not sure how to determine which files RPM thinks are config files? Chapter 5, specifically, section 5.2.2 on page 61 will show you how.

3.5 Watch Out!

RPM takes most of the work out of removing software from your system, and that's great. As with everything else in life, however, there's a downside. RPM also makes it easy to erase packages that are critical to your system's continued operation. Here are some examples of packages *not* to erase:

- RPM: RPM will happily uninstall itself. No problem — you'll just re-install it with `rpm -i`... Oops!
- Bash: The Bourne-again Shell may not be the shell you use, but certain parts of many Linux systems (like the scripts executed during system startup and shutdown) use `/bin/sh`, which is a symbolic link to `/bin/bash`. No `/bin/bash`, no `/bin/sh`. No `/bin/sh`, no system!

In many cases, RPM's dependency processing will prevent inadvertent erasures from causing massive problems. However, if you're not sure, use `rpm -q` to get more information about the package you'd like to erase.[2]

[2] See chapter 5 on page 51 for more information on `rpm -q`.

Chapter 4

Using RPM to Upgrade Packages

`rpm -U` (or `--upgrade`) *options* file1.rpm ... fileN.rpm		
Parameters		
`file1.rpm ... fileN.rpm`	One or more RPM package files (URLs OK)	
Upgrade—specific Options		Page
`-h` (or `--hash`)	Print hash marks ("`#`") during upgrade‡	20
`--oldpackage`	Permit "upgrading" to an older package	47
`--test`	Perform upgrade tests only‡	21
`--excludedocs`	Do not install documentation‡	28
`--includedocs`	Install documentation‡	29
`--replacepkgs`	Replace a package with a new copy of itself‡	22
`--replacefiles`	Replace files owned by another package‡	23
`--force`	Ignore package and file conflicts	48
`--percent`	Print percentages during upgrade‡	31
`--noscripts`	Do not execute pre- and post-install scripts	48
`--prefix` *<path>*	Relocate package to *<path>* if possible‡	30
`--ignorearch`	Do not verify package architecture‡	33
`--ignoreos`	Do not verify package operating system‡	34
`--nodeps`	Do not check dependencies‡	27
`--ftpproxy` *<host>*	Use *<host>* as the FTP proxy‡	33
`--ftpport` *<port>*	Use *<port>* as the FTP port‡	33
General Options		Page
`-v`	Display additional information‡	19
`-vv`	Display debugging information‡	21
`--root` *<path>*	Set alternate root to *<path>*‡	32
`--rcfile` *<rcfile>*	Set alternate rpmrc file to *<rcfile>*‡	32
`--dbpath` *<path>*	Use *<path>* to find the RPM database‡	32

‡This option behaves identically to the same option used with `rpm -i`. Please see Chapter 2 for more information on this option.

4.1 rpm -U — What Does it Do?

If there was one RPM command that could win over friends, it would be RPM's upgrade command. After all, anyone who has ever tried to install a newer version of *any* software knows what a traumatic experience it can be. With RPM, though, this process is reduced to a single command: `rpm -U`. The `rpm -U` command (`--upgrade` is equivalent) performs two distinct operations:

1. Installs the desired package.
2. Erases all older versions of the package, if any exist.

If it sounds to you like `rpm -U` is nothing more than an `rpm -i` command (see chapter 2) followed by the appropriate number of `rpm -e` commands, (see chapter 3) you'd be exactly right. In fact, we'll be referring back to those chapters as we discuss `rpm -U`, so if you haven't skimmed those chapters yet, you might want to do that now.

While some people might think it's a "cheap shot" to claim that RPM performs an upgrade when in fact it's just doing the equivalent of a couple of other commands, in fact, it's a very smart thing to do. By carefully crafting RPM's package installation and erasure commands to do the work required during an upgrade, it makes RPM more tolerant of misuse by preserving important files even if an upgrade isn't being done.

If RPM had been written with a very "smart" upgrade command, and the install and erase commands couldn't handle upgrade situations at all, installing a package could overwrite a modified configuration file. Likewise, erasing a package would also mean that config files could be erased. Not a good situation! However, RPM's approach to upgrades makes it possible to handle even the most tricky situation — having multiple versions of a package install simultaneously.

4.1.1 Config file magic

While the `rpm -i` and `rpm -e` commands each do their part to keep config files straight, it is with `rpm -U` that the full power of RPM's config file handling shows through. There are no less than *six* different scenarios that RPM takes into account when handling config files.

In order to make the appropriate decisions, RPM needs information. The information used to decide how to handle config files is a set of three large numbers known as *MD5 checksums*. An MD5 checksum is produced when a file is used as the input to a complex series of mathematical operations. The resulting checksum has a unique property, in that *any* change to the file's contents will result in a change to the checksum of that file.[1] Therefore, MD5 checksums are a powerful tool for quickly determining whether two different files have the same contents or not.

In the previous paragraph, we stated that RPM uses three different MD5 checksums to determine what should be done with a config file. The three checksums are:

[1] Actually, there's a one in 2^{128} chance a change will go undetected, but for all practical purposes, it's as close to perfect as we can get.

1. The MD5 checksum of the file when it was originally installed. We'll call this the *original file*.
2. The MD5 checksum of the file as it exists at upgrade time. We'll call this the *current file*.
3. The MD5 checksum of the corresponding file in the new package. We'll call this the *new file*.

Let's take a look at the various combinations of checksums, see what RPM will do because of them, and discuss why. In the following examples, we'll use the letters X, Y, and Z in place of lengthy MD5 checksums.

Original file = X, Current file = X, New file = X – In this case, the file originally installed was never modified.[2] The file in the new version of the package is identical to the file on disk.

In this case, RPM installs the new file, overwriting the original. You may be wondering why go to the trouble of installing the new file if it's just the same as the existing one. The reason is that aspects of the file *other* than its name and contents might have changed. The file's ownership, for example, might be different in the new version.

Original file = X, Current file = X, New file = Y – The original file has not been modified, but the file in the new package *is* different. Perhaps the difference represents a bug-fix, or a new feature. It makes no difference to RPM.

In this case, RPM installs the new file, overwriting the original. This makes sense. If it didn't, RPM would never permit newer, modified versions of software to be installed! The original file is not saved, since it had not been changed. A lack of changes here means that no site-specific modifications were made to the file.

Original file = X, Current file = Y, New file = X – Here we have a file that *was* changed at some point. However, the new file is identical to the existing file *prior* to the local modifications.

In this case, RPM takes the viewpoint that since the original file and the new file are identical, the modifications made to the original version must still be valid for the new version. It leaves the existing, modified file in place.

Original file = X, Current file = Y, New file = Y – At some point the original file was modified, and those modifications happen to make the file identical to the new file. Perhaps the modification was made to fix a security problem, and the new version of the file has the same fix applied to it.

[2] Or, as some sticklers for detail may note, it may have been modified, and subsequently those modifications were undone.

In this case, RPM installs the new version, overwriting the modified original. The same philosophy used in the first scenario applies here – although the file has not changed, perhaps some other aspect of the file has, so the new version is installed.

Original file = X, Current file = Y, New file = Z – Here the original file was modified at some point. The new file is different from both the original *and* the modified versions of the original file.

RPM is not able to analyze the contents of the files, and determine what is going on. In this instance, it takes the best possible approach. The new file is known to work properly with the rest of the software in the new package — at least the people building the new package should have insured that it does. The modified original file is an unknown: it might work with the new package, it might not. So RPM installs the new file.

BUT... The existing file was definitely modified. Someone made an effort to change the file, for some reason. Perhaps the information contained in the file is still of use. Therefore, RPM saves the modified file, naming it *file*.rpmsave, and prints a warning, so the user knows what happened:

```
warning: /etc/skel/.bashrc saved as /etc/skel/.bashrc.rpmsave
```

These five scenarios cover just about every possible circumstance, save one. The missing scenario?

Original file = *none*, Current file = ??, New file = ?? – While RPM doesn't use checksums in this particular case, we'll describe it in those terms, for the sake of consistency. In this instance, RPM had not installed the file originally, so there is no original checksum.

Because the file had not originally been installed as part of a package, there is no way for RPM to determine if the file currently in place had been modified. Therefore, the checksums for the current file and the new file are irrelevant; they cannot be used to clear up the mystery.

When this happens, RPM renames the file to *file*.rpmorig, prints a warning, and installs the new file. This way, any modifications contained in the original file are saved. The system administrator can review the differences between the original and the newly installed files and determine what action should be taken.

As you can see, in the majority of cases RPM will automatically take the proper course of action when performing an upgrade. It is only when config files have been modified and are to be overwritten, that RPM leaves any post-upgrade work for the system administrator. Even in those cases, many times the modified files are not worth saving and can be deleted.

4.2 Upgrading a Package

The most basic version of the `rpm -U` command is simply "`rpm -U`", followed by the name of a .rpm package file:

```
# rpm -U eject-1.2-2.i386.rpm
#
```

Here, RPM performed all the steps necessary to upgrade the `eject-1.2-2` package, faster than could have been done by hand. As in RPM's install command, Uniform Resource Locators, or URLs, can also be used to specify the package file.[3]

4.2.1 `rpm -U`'s Dirty Little Secret

Well, in the example above, we didn't tell the whole story. There was no older version of the `eject` package installed. Yes, it's true — `rpm -U` works just fine as a replacement for the normal install command `rpm -i`.

This is another, more concrete example of the strength of RPM's method of performing upgrades. Since RPM's install command is smart enough to handle upgrades, RPM's upgrade command is really just another way to specify an install. Some people never even bother to use RPM's install command; they always use `rpm -U`. Maybe the "`-U`" should stand for, "Uh, do the right thing"...

4.3 They're Nearly Identical...

Given the fact that `rpm -U` can be used as a replacement to `rpm -i`, it follows that most of the options available for `rpm -U` are identical to those used with `rpm -i`. Therefore, to keep the duplication to a minimum, we'll discuss only those options that are unique to `rpm -U`, or that behave differently from the same option when used with `rpm -i`. The table on page 43 at the start of this chapter shows all valid options to RPM's upgrade command, and indicates which are identical to those used with `rpm -i`.

4.3.1 `--oldpackage`: Upgrade To An Older Version

This option might be used a bit more by people that like to stay on the "bleeding edge" of new versions of software, but eventually, everyone will probably need to use it. Usually, the situation plays out like this:

- You hear about some new software that sounds pretty nifty, so you download the .rpm file and install it.

[3] For more information on RPM's use of URLs, please see section 2.2.1 on page 17.

- The software is *great*! It does everything you ask for, and more. You end up using it every day for the next few months.
- You hear that a new version of your favorite software is available. You waste no time in getting the package. You upgrade the software by using `rpm -U`. No problem!
- Fingers arched in anticipation, you launch the new version. Your computer's screen goes blank!

Looks like a bug in the new version. *Now* what do you do? Hmmm. Maybe you can just "upgrade" to the older version. Let's try to go back to release 2 of `cdp-0.33` from release 3:

```
# rpm -Uv cdp-0.33-2.i386.rpm
Installing cdp-0.33-2.i386.rpm
package cdp-0.33-3 (which is newer) is already installed
error:  cdp-0.33-2.i386.rpm cannot be installed
#
```

That didn't work very well. At least it told us just what the problem was — we were trying to upgrade to an older version of a package that is already installed. Fortunately, there's a special option for just this situation: `--oldpackage`. Let's give it a try:

```
# rpm -Uv --oldpackage cdp-0.33-2.i386.rpm
Installing cdp-0.33-2.i386.rpm
```

By using the `--oldpackage` option, release 3 of `cdp-0.33` is history, and has been replaced by release 2.

4.3.2 `--force`: The Big Hammer

Adding `--force` to an upgrade command is a way of saying "Upgrade it anyway!" In essence, it adds `--replacepkgs`, `--replacefiles`, *and* `--oldpackage` to the command. Like a big hammer, `--force` is an irresistible force[4] that makes things happen. In fact, the only thing that will prevent a `--force`'ed upgrade from proceeding is a dependency conflict.

And like a big hammer, it pays to fully understand why you need to use `--force` before actually using it.

4.3.3 `--noscripts`: Do Not Execute Install and Uninstall Scripts

The `--noscripts` option prevents a package's pre- and post-install scripts from being executed. This is no different than the option's behavior when used with RPM's install com-

[4] Pun intended.

mand. However, there is an additional point to consider when the option is used during an upgrade. The following example uses specially-built packages that display messages when their scripts are executed by RPM:

```
# rpm -i bother-2.7-1.i386.rpm
This is the bother 2.7 preinstall script
This is the bother 2.7 postinstall script
#
```

In this case, a package has been installed. As expected, its scripts are executed. Next, let's upgrade this package:

```
# rpm -U bother-3.5-1.i386.rpm
This is the bother 3.5 preinstall script
This is the bother 3.5 postinstall script
This is the bother 2.7 preuninstall script
This is the bother 2.7 postuninstall script
#
```

This is a textbook example of the sequence of events during an upgrade. The new version of the package is installed (as shown by the pre- and post-install scripts being executed). Finally, the previous version of the package is removed (showing the pre- and post-*un*install scripts being executed).

There are really no surprises there — it worked just the way it was meant to. This time, let's use the `--noscripts` option when the time comes to perform the upgrade:

```
# rpm -i bother-2.7-1.i386.rpm
This is the bother 2.7 preinstall script
This is the bother 2.7 postinstall script
```

Again, the first package is installed, and its scripts are executed. Now let's try the upgrade using the `--noscripts` option:

```
# rpm -U --noscripts bother-3.5-1.i386.rpm
This is the bother 2.7 preuninstall script
This is the bother 2.7 postuninstall script
#
```

The difference here is that the `--noscripts` option prevented the new package's scripts from executing. The scripts from the package being erased were still executed.

Chapter 5

Getting Information About Packages

rpm -q (or `--query`) *options*

Package Selection Options		Page
`pkg1 ... pkgN`	Query installed package(s)	52
`-p` *<file>* (or \`\`-'')	Query package file *<file>* (URLs OK)	56
`-f` *<file>*	Query package owning *<file>*	54
`-a`	Query all installed packages	54
`--whatprovides` *<x>*	Query packages providing capability *<x>*	58
`-g` *<group>*	Query packages belonging to group *<group>*	57
`--whatrequires` *<x>*	Query packages requiring capability *<x>*	58
Information Selection Options		Page
<null>	Display full package label	52
`-i`	Display summary package information	59
`-l`	Display list of files in package	60
`-c`	Display list of configuration files	61
`-d`	Display list of documentation files	62
`-s`	Display list of files in package, with state	62
`--scripts`	Display install, uninstall, verify scripts	66
`--queryformat (or --qf)`	Display queried data in custom format	67
`--dump`	Display all verifiable information for each file	65
`--provides`	Display capabilities package provides	63
`--requires (or -R)`	Display capabilities package requires	64
General Options		Page
`-v`	Display additional information	61
`-vv`	Display debugging information	74
`--root` *<path>*	Set alternate root to *<path>*	74
`--rcfile` *<rcfile>*	Set alternate `rpmrc` file to *<rcfile>*	75
`--dbpath` *<path>*	Use *<path>* to find the RPM database	75

5.1 `rpm -q` — What does it do?

One of the nice things about using RPM is that the packages you manage don't end up going into some kind of black hole. Nothing would be worse than to install, upgrade, and erase several different packages and not have a clue as to what's on your system. In fact, RPM's query function can help you get out of sticky situations like:

- You're poking around your system, and you come across a file that you just can't identify. Where did it come from?
- Your friend sends you a package file, and you have no idea what the package does, what it installs, or where it originally came from.
- You know that you installed XFree86 a couple months ago, but you don't know what version, and you can't find any documentation on it.

The list could go on, but you get the idea. The `rpm -q` command is what you need. If you're the kind of person that doesn't like to have more options than you know what to do with, `rpm -q` might look imposing. But fear not. Once you have a handle on the basic structure of an RPM query, it'll be a piece of cake.

5.2 The Parts of an RPM Query

It becomes easy to construct a query command once you understand the individual parts. First is the `-q` (You can also use `--query`, if you like). After all, you need to tell RPM what function to perform, right? The rest of a query command consists of two distinct parts: package selection (or what packages you'd like to query), and information selection (or what information you'd like to see). Let's take a look at package selection first:

5.2.1 Query Commands, Part One: Package Selection

The first thing you'll need to decide when issuing an RPM query is what package (or packages) you'd like to query. RPM has several ways to specify packages, so you have quite an assortment to choose from.

The Package Label

In earlier chapters, we discussed RPM's package label, a string that uniquely identifies every installed package. Every label contains three pieces of information:

1. The *name* of the packaged software.
2. The *version* of the packaged software.
3. The package's *release* number.

When issuing a query command using package labels, you must always include the package name. You can also include the version and even the release, if you like. The only restrictions are that each part of the package label specified must be complete, and that if any parts of the package label are missing, all parts to the right must be omitted as well. This second restriction is just a long way of saying that if you specify the release, you must also specify the version as well. Let's look at a few examples.

Say, for instance, you've recently installed a new version of the C libraries, but you can't remember the version number:

```
# rpm -q libc
libc-5.2.18-1
#
```

In this type of query, RPM returns the complete package label for all installed packages that match the given information. In the example above, if version 5.2.17 of the C libraries was also installed, its package label would have been displayed, too.

In this example, we've included the version as well as the package name:

```
# rpm -q rpm-2.3
rpm-2.3-1
#
```

Note, however, that RPM is a bit picky about specifying package names. Here are some queries for the C library that *won't* work:

```
# rpm -q LibC
package LibC is not installed
#

# rpm -q lib
package lib is not installed
#

# rpm -q "lib*"
package lib* is not installed
#

# rpm -q libc-5
package libc-5 is not installed
#

# rpm -q libc-5.2.1
package libc-5.2.1 is not installed
#
```

As you can see, RPM is case sensitive about package names and cannot match partial names, version numbers, or release numbers. Nor can it use the wildcard characters we've come to know and love. As we've seen, however, RPM can perform the query when more than one field of the package label is present. In the above case, `rpm -q libc-5.2.18`, or even `rpm -q libc-5.2.18-1` would have found the package, `libc-5.2.18-1`.

Querying based on package labels may seem a bit restrictive. After all, you need to know the exact name of a package in order to perform a query on it. But there are other ways of specifying packages...

`-a` — Query All Installed Packages

Want lots of information fast? Using the `-a` option, you can query every package installed on your system. For example:

```
# rpm -qa
ElectricFence-2.0.5-2
ImageMagick-3.7-2
...
tetex-xtexsh-0.3.3-8
lout-3.06-4
#
```

(On a system installed using RPM, the number of packages can easily number 200 or more; we've deleted most of the output.)

The `-a` option can produce mountains of output, which makes it a prime candidate for piping through the many Linux/UNIX commands available. One of the prime candidates would be a pager such as `more`, so that the list of installed packages could be viewed a screenful at a time.

Another handy command to pipe `rpm -qa`'s output through is `grep`. In fact, using `grep`, it's possible to get around RPM's lack of built-in wildcard processing:

```
# rpm -qa | grep -i sysv
SysVinit-2.64-2
#
```

In this example, we were able to find the `SysVinit` package, even though we didn't have the complete package name, or capitalization.

`-f` *<file>* — Query the Package Owning *<file>*

How many times have you found a program sitting on your system and wondered "what does it do?" Well, if the program was installed by RPM as part of a package, it's easy to find out. Simply use the `-f` option. Example: You find a strange program called `ls` in `/bin` (Okay, it *is* a contrived example). Wonder what package installed it? Simple!

```
# rpm -qf /bin/ls
fileutils-3.12-3
#
```

If you happen to point RPM at a file it didn't install, you'll get a message similar to the following:

```
# rpm -qf .cshrc
file /home/ed/.cshrc is not owned by any package
#
```

A Tricky Detail It's possible that you'll get the "not owned by any package" message in error. Here's an example of how it can happen:

```
# rpm -qf /usr/X11/bin/xterm
file /usr/X11/bin/xterm is not owned by any package
#
```

As you can see, we're trying to find out what package the `xterm` program is part of. The first example failed, which might lead one to believe that `xterm` really *isn't* owned by any package.

However, let's look at a directory listing:

```
# ls -lF /usr
...
lrwxrwxrwx 1 root root 5 May 13 12:46 X11 -> X11R6/
drwxrwxr-x 7 root root 1024 Mar 21 00:21 X11R6/
...
#
```

(We've truncated the list; normally `/usr` is quite a bit more crowded than this.)

The key here is the line ending with "`X11 -> X11R6/`". This is known as a "symbolic link". It's a way of referring to a file (here, a directory file) by another name. In this case, if we used the path `/usr/X11`, or `/usr/X11R6`, it shouldn't make a difference. It certainly doesn't make a difference to programs that simply want access to the file. But it does make a difference to RPM, because RPM doesn't use the filename to access the file. RPM uses the filename as a key into its database. It would be very difficult, if not impossible, to keep track of all the symlinks on a system and try every possible path to a file during a query.

What to do? There are two options:

1. Make sure you always specify a path free of symlinks. This can be pretty tough, though. An alternative approach is to use `namei` to track down symlinks:

```
# namei /usr/X11/bin/xterm
f:  /usr/X11/bin/xterm
 d /
 d usr
 l X11 -> X11R6
   d X11R6
 d bin
 - xterm
#
```

   It's pretty easy to see the `X11` to `X11R6` symlink. Using this approach you can enter the non-symlinked path and get the desired results:

```
# rpm -qf /usr/X11R6/bin/xterm
XFree86-3.1.2-5
#
```

2. Change your directory to the one holding the file you want to query. Even if you use a symlinked path to get there, querying the file should then work as you'd expect:

```
# cd /usr/X11/bin
# rpm -qf xterm
XFree86-3.1.2-5
#
```

So if you get a "not owned by any package" error, and you think it may not be true, try one of the approaches above.

`-p <file>` — Query a Specific RPM Package File

Up to now, every means of specifying a package to an RPM query focused on packages that had already been installed. While it's certainly very useful to be able to dredge up information about packages that are already on your system, what about packages that haven't yet been installed? The `-p` option can do that for you.

One situation where this capability would help, occurs when the name of a package file has been changed. Since the name of the file containing a package has *nothing* to do with the name of the package (though, by tradition it's nice to name package files consistently), we can use this option to find out exactly what package a file contains:

```
# rpm -qp foo.bar
rpm-2.3-1
#
```

With one command RPM gives you the answer.[1]

The `-p` option can also use Uniform Resource Locators to specify package files. See section 2.2.1 on page 17 for more information on using URLs.

There's one last trick up `-p`'s sleeve — it can also perform a query by reading a package from standard input. Here's an example:

```
# cat bother-3.5-1.i386.rpm | rpm -qp -
bother-3.5-1
#
```

We piped the output of `cat` into RPM. The dash at the end of the command line directs RPM to read the package from standard input.

-g *<group>*: Query Packages Belonging To Group *<group>*

When a package is built, the package builder must classify the package, grouping it with other packages that perform similar functions. RPM gives you the ability to query installed packages based on their groups. For example, there is a group known as `Base`. This group consists of packages that provide low-level structure for a Linux distribution. Let's see what installed packages make up the `Base` group:

```
# rpm -qg Base
setup-1.5-1
pamconfig-0.50-5
filesystem-1.2-1
crontabs-1.3-1
dev-2.3-1
etcskel-1.1-1
initscripts-2.73-1
mailcap-1.0-3
pam-0.50-17
passwd-0.50-2
redhat-release-4.0-1
rootfiles-1.3-1
termcap-9.12.6-5
#
```

[1] On most Linux systems, the `file` command can be used to obtain similar information. See Appendix A for details on how to add this capability to your system's `file` command.

One thing to keep in mind is that group specifications are case-sensitive. Issuing the command `rpm -qg base` won't produce any output.

`--whatprovides <x>`: Query the Packages That Provide Capability `<x>`

RPM provides extensive support for dependencies between packages. The basic mechanism used is that a package may *require* what another package *provides*. The thing that is required and provided can be a shared library's soname. It can also be a character string chosen by the package builder. In any case, it's important to be able to display which packages provide a given capability.

This is just what the `--whatprovides` option does. When the option, followed by a capability, is added to a query command, RPM will select those packages that provide the capability. Here's an example:

```
# rpm -q --whatprovides module-info
kernel-2.0.18-5
#
```

In this case, the only package that provides the `module-info` capability is `kernel-2.0.18-5`.

`--whatrequires <x>`: Query the Packages That Require capability `<x>`

The `--whatrequires` option is the logical complement to the `--whatprovides` option described above. It is used to display which packages require the specified capability. Expanding on the example we started with `--whatprovides`, let's see which packages require the `module-info` capability:

```
# rpm -q --whatrequires module-info
kernelcfg-0.3-2
#
```

There's only one package that requires `module-info` — `kernelcfg-0.3-2`.

5.2.2 Query Commands, Part Two: Information Selection

After specifying the package (or packages) you wish to query, you'll need to figure out just *what* information you'd like RPM to retrieve. As we've seen, by default, RPM only returns the complete package label. But there's much more to a package than that. Here, we'll explore every information selection option available to us.

`-i` — Display Package Information

Adding `-i` to `rpm -q` tells RPM to give you some information on the package or packages you've selected. For the sake of clarity, let's take a look at what it gives you and explain what you're looking at:

```
# rpm -qi rpm
Name        : rpm                       Distribution: Red Hat Linux Vanderbilt
Version     : 2.3                             Vendor: Red Hat Software
Release     : 1                           Build Date: Tue Dec 24 09:07:59 1996
Install date: Thu Dec 26 23:01:51 1996 Build Host: porky.redhat.com
Group       : Utilities/System            Source RPM: rpm-2.3-1.src.rpm
Size        : 631157
Summary     : Red Hat Package Manager
Description :
RPM is a powerful package manager, which can be used to build, install,
query, verify, update, and uninstall individual software packages. A
package consists of an archive of files, and package information, including
name, version, and description.
#
```

There's quite a bit of information here, so let's go through it entry by entry:

- `Name` — The name of the package you queried. Usually (but not always) it bears some resemblance to the name of the underlying software.
- `Version` — The version number of the software, as specified by the software's original creator.
- `Release` — The number of times a package consisting of this software has been packaged. If the version number should change, the release number should start over again at "1".

As you've probably noticed, these three pieces of information comprise the package label we've come to know and love. Continuing, we have:

- `Install date` — This is the time when the package was installed on your system.
- `Group` — In our example, this looks suspiciously like a path. If you went searching madly for a directory tree by that name, you'd come up dry — it isn't a set of directories at all.

  When a package builder starts to create a new package, they enter a list of words that describe the software. The list, which goes from least specific to most specific, attempts to categorize the software in a concise manner. The primary use for the group is to enable graphically oriented package managers based on RPM to present packages grouped by function. Red Hat Linux's `glint` command does this.

- `Size` — This is the size (in bytes) of every file in this package. It might make your decision to erase an unused package easier if you see six or more digits here.

- `Summary` — This is a concise description of the packaged software.

- `Description` — This is a verbose description of the packaged software. Some descriptions might be more, well, *descriptive* than others, but hopefully it will be enough to clue you in as to the software's role in the greater scheme of things.

- `Distribution` — The word "distribution" is really not the best name for this field. "Product" might be a better choice. In any case, this is the name of the product this package is a part of.

- `Vendor` — The organization responsible for building this package.

- `Build Date` — The time the package was created.

- `Build Host` — The name of the computer system that built the package.[2]

- `Source RPM` — The process of building a package results in two files:

  1. The package file used to install the packaged software. This is sometimes called the *binary* package.
  2. The package file containing the source code and other files used to create the binary package file. This is known as the *source* RPM package file. This is the filename that is displayed in this field.

  Unless you want to make changes to the software, you probably won't need to worry about source packages. But if you do, stick around, because the second part of this book is for you...

`-l` — Display the Package's File List

Adding `-l` to `rpm -q` tells RPM to display the list of files that are installed by the specified package or packages. If you've used `ls` before, you won't be surprised by RPM's file list.

Here's a look at one of the smaller packages on Red Hat Linux — `adduser`:

```
# rpm -ql adduser
/usr/sbin/adduser
#
```

The `adduser` package consists of only one file, so there's only one filename displayed.

[2] Note to software packagers: Choose your build machine names wisely! A silly or offensive name might be embarrassing...

`-v` — **Display Additional Information** In some cases, the `-v` option can be added to a query command for additional information. The `-l` option we've been discussing is an example of just such a case. Note how the `-v` option adds verbosity:

```
# rpm -qlv adduser
-rwxr-xr-x- root root 3894 Feb 25 13:45 /usr/sbin/adduser
#
```

Looks a lot like the output from `ls`, doesn't it? Looks can be deceiving. Everything you see here is straight from RPM's database. However, the format is identical to `ls`, so it's more easily understood. If this is Greek to you, consult the `ls` man page.

`-c` — Display the Package's List of Configuration Files

When `-c` is added to an `rpm -q` command, RPM will display the configuration files that are part of the specified package or packages. As mentioned earlier in the book, config files are important, because they control the behavior of the packaged software. Let's take a look at the list of config files for `XFree86`:

```
# rpm -qc XFree86
/etc/X11/fs/config
/etc/X11/twm/system.twmrc
/etc/X11/xdm/GiveConsole
/etc/X11/xdm/TakeConsole
/etc/X11/xdm/Xaccess
/etc/X11/xdm/Xresources
/etc/X11/xdm/Xservers
/etc/X11/xdm/Xsession
/etc/X11/xdm/Xsetup_0
/etc/X11/xdm/chooser
/etc/X11/xdm/xdm-config
/etc/X11/xinit/xinitrc
/etc/X11/xsm/system.xsm
/usr/X11R6/lib/X11/XF86Config
#
```

These are the files you'd want to look at first if you were looking to customize XFree86 for your particular needs. Just like `-l`, we can also add `v` for more information:

```
# rpm -qcv XFree86
-r--r--r---  root  root    423 Mar 21 00:17 /etc/X11/fs/config
...
lrwxrwxrwx-  root  root     30 Mar 21 00:29 /usr/X11R6/lib/X11/XF86Config
-> ../../../../etc/X11/XF86Config
#
```

(Note that last file: RPM will display symbolic links, as well.)

-d — Display a List of the Package's Documentation

When `-d` is added to a query, we get a list of all files containing documentation for the named package or packages. This is a great way to get up to speed when you're having problems with unfamiliar software. As with `-c` and `-l`, you'll see either a simple list of filenames, or (if you've added `-v`) a more comprehensive list. Here's an example that might look daunting at first, but really isn't:

```
# rpm -qdcf /sbin/dump
/etc/dumpdates
/usr/doc/dump-0.3-5
/usr/doc/dump-0.3-5/CHANGES
/usr/doc/dump-0.3-5/COPYRIGHT
/usr/doc/dump-0.3-5/INSTALL
/usr/doc/dump-0.3-5/KNOWNBUGS
/usr/doc/dump-0.3-5/THANKS
/usr/doc/dump-0.3-5/dump-0.3.announce
/usr/doc/dump-0.3-5/dump.lsm
/usr/doc/dump-0.3-5/linux-1.2.x.patch
/usr/man/man8/dump.8
/usr/man/man8/rdump.8
/usr/man/man8/restore.8
/usr/man/man8/rmt.8
/usr/man/man8/rrestore.8
#
```

Let's take that alphabet soup set of options, one letter at a time:

- `q` — Perform a query.
- `d` — List all documentation files.
- `c` — List all config files.
- `f` — Query the package that owns the specified file (`/sbin/dump`, in this case).

The list of files represents all the documentation and config files that apply to the package owning `/sbin/dump`.

-s — Display the State of Each File in the Package

Unlike the past three sections, which dealt with a list of files of one type or another, adding `-s` to a query will list the *state* of the files that comprise one or more packages. I can hear you out there; you're saying, "What is the *state* of a file?" For every file that RPM installs, there is an associated state. There are four possible states:

1. `normal` — A file in the `normal` state has not been modified by installing another package on the system.

2. `replaced` — Files in the `replaced` state have been modified by installing another package on the system.

3. `not installed` — A file is classified as `not installed` when it, er, isn't installed! This state is normally seen only if the package was partially installed. An example of a partially installed package would be one that was installed with the `--excludedocs` option. Using this option, no documentation files would be installed. The RPM database would still contain entries for these missing files, but their state would be `not installed`.

4. `net shared` — The `net shared` state is used to support client systems that NFS mount portions of their filesystems from a server. Since the server most likely exports filesystems to more than one client, if a client erased a package that contained files on a shared filesystem, other client systems would have incompletely installed packages. The `net shared` state is used to alert RPM to the fact that a file is on a shared filesystem and should not be erased. Files will be in the `net shared` state when two things happen:

   (a) The `netsharedpath` `rpmrc` file entry has been changed from its default (null) value.[3]

   (b) The file is to be installed in a directory within a net shared path.

Here's an example showing how file states appear:

```
# rpm -qs adduser
normal /usr/sbin/adduser
#
```

(That `normal` at the start of the line is the state, followed by the file name)

The file state is one of the tools RPM uses to determine the most appropriate action to take when packages are installed or erased.

Now would the average person need to check the states of files? Not really. But if there should be problems, this kind of information can help get things back on track.

`--provides`: Display Capabilities Provided by the Package

By adding `--provides` to a query command, we can see the capabilities provided by one or more packages. If the package doesn't provide any capabilities, the `--provides` option produces no output:

```
# rpm -q --provides rpm
```

[3]For more information on `rpmrc` file entries, please refer to Appendix B on page 363.

```
#
```

However, if a package *does* provide capabilities, they will be displayed:

```
# rpm -q --provides foonly
index
#
```

It's important to remember that capabilities are *not* filenames. In the above example, the `foonly` package contains no file called `index`; it's just a character string the package builder chose. This is no different from the following example:

```
# rpm -q --provides libc
libm.so.5
libc.so.5
#
```

While there might be symlinks by those names in `/lib`, capabilities are a property of the *package*, not a file contained in the package!

`--requires`: Display Capabilities Required by the Package

The `--requires` option (`-R` is equivalent) is the logical complement to the `--provides` option. It displays the capabilities required by the specified package(s). If a package has no requirements, there's no output:

```
# rpm -q --requires adduser
#
```

In cases where there *are* requirements, they are displayed as follows:

```
# rpm -q --requires rpm
libz.so.1
libdb.so.2
libc.so.5
#
```

It's also possible that you'll come across something like this:

```
# rpm -q --requires blather
bother >= 3.1
#
```

Packages may also be built to require another package. This requirement can also include specific versions. In the example above, the `bother` package is required by `blather`; specifically, a version of `bother` greater than or equal to 3.1.

Here's something worth understanding. Let's say we decide to track down the `bother` that `blather` says it requires. If we use RPM's query capabilities, we could use the `--whatprovides` package selection option to try to find it:

```
# rpm -q --whatprovides bother
no package provides bother
#
```

No dice. This might lead you to believe that the `blather` package has a problem. The moral of this story is that, when trying to find out what package fulfills another package's requirements, it's a good idea to also try a simple query using the requirement as a package name. Continuing our example above, let's see if there's a package called `bother`:

```
# rpm -q bother
bother-3.5-1
#
```

Bingo! However, if we see what capabilities the `bother` package provides, we come up dry:

```
# rpm -q --provides bother
#
```

The reason for the lack of output is that all packages, by default, "provide" their package name (and version).

`--dump`: Display All Verifiable Information for Each File

The `--dump` option is used to display every piece of information RPM has on one or more files listed in its database. The information is listed in a very concise fashion. Since the `--dump` option displays file-related information, the list of files must be chosen by using the `-l`, `-c`, or `-d` options (or some combination thereof):

```
# rpm -ql --dump adduser
/usr/sbin/adduser 4442 841083888 ca5fa53dc74952aa5b5e3a5fa5d8904b 0100755
root root 0 0 0 X
#
```

What does all this stuff mean? Let's go through it, item-by-item:

- The `/usr/sbin/adduser` is simple: it's the name of the file being `dump`'ed.
- `4442` is the size of the file, in bytes.
- How about `841083888`? It's the time the file was last modified, in seconds past the Unix zero date of January 1, 1970.
- The `ca5fa53dc74952aa5b5e3a5fa5d8904b` is the MD5 checksum of the file's contents, all 128 bits of it.
- If you guessed `0100755` was the file's mode, you'd be right.
- The first `root` represents the file's owner.
- The second `root` is the file's group.
- We'll take the next part (`0 0`) in one chunk. The first zero shows whether the file is a config file. If zero, as in this case, then the file is not a config file. The next zero shows whether the file is documentation. Again, since there is a zero here, this file isn't documentation, either.
- The final `0` represents the file's major and minor numbers. These are set only for device special files. Otherwise, it will be zero.
- If the file were a symlink, the spot taken by the `X` would contain a path pointing to the linked file.

Normally, the `--dump` option is used by people that want to extract the file-related information from RPM and process it somehow.

--scripts — Show Scripts Associated With a Package

If you add `--scripts` (that's *two* dashes) to a query, you get to see a little bit more of RPM's underlying magic:

```
# rpm -q --scripts XFree86
preinstall script:
(none)

postinstall script:
/sbin/ldconfig
/sbin/pamconfig --add --service=xdm --password=none --sesslist=none


preuninstall script:
(none)

postuninstall script:
/sbin/ldconfig
```

```
if [ "$1" = 0 ] ; then
/sbin/pamconfig --remove --service=xdm --password=none --sesslist=none
fi


verify script:
(none)

#
```

In this particular case, the `XFree86` package has two scripts: one labeled `postinstall`, and one labeled `postuninstall`. As you might imagine, the postinstall script is executed just after the package's files have been installed; the postuninstall script is executed just after the package's files have been erased.

Based on the labels in this example, you'd probably imagine that a package can have as many as five different scripts. You'd be right:

1. The preinstall script, which is executed just *before* the package's files are installed.
2. The postinstall script, which is executed just *after* the package's files are installed.
3. The preuninstall script, which is executed just *before* the package's files are removed.
4. The postuninstall script, which is executed just *after* the package's files are removed.
5. And finally, the verify script. While it's easy to figure out the other scripts' functions based on their name, what does a script called *verify* do? Well, we haven't gotten to it yet, but packages can also be verified for proper installation. This script is used during verification.[4]

Is this something you'll need very often? As in the case of displaying file states, not really. But when you need it, you *really* need it!

`--queryformat` - Construct a Custom Query Response

OK, say you're *still* not satisfied. You'd like some additional information, or you think a different format would be easier on the eyes. Maybe you want to take some information on the packages you've installed and run it through a script for some specialized processing. You can do it, using the `--queryformat` option. In fact, if you look back at the output of the `-i` option, RPM was using `--queryformat` internally. Here's how it works:

On the RPM command line, include `--queryformat`. Right after that, enter a format string, enclosed in single quotes "'".

The format string can consist of a number of different components:

- Literal text, including escape sequences.

[4] For more information on package verification, please see section 6.1 on page 79.

- Tags, with optional field width, formatting, and iteration information.
- Array Iterators.

Let's look at each of these components.

Literal text Any part of a format string that is not associated with tags or array iterators will be treated as literal text. Literal text is just that: It's text that is printed just as it appears in the format string. In fact, a format string can consist of nothing but literal text, although the output wouldn't tell us much about the packages being queried. Let's give the `--queryformat` option a try, using a format string with nothing but literal text:

```
# rpm -q --queryformat 'This is a test!' rpm
This is a test!#
```

The RPM command might look a little unusual, but if you take out the `--queryformat` option, along with its format string, you'll see this is just an ordinary query of the `rpm` package. When the `--queryformat` option is present, RPM will use the text immediately following the option as a format string. In our case, the format string is `'This is a test!'`. The single quotes are required. Otherwise, it's likely your shell will complain about some of the characters contained in the average format string.

The output of this command appears on the second line. As we can see, the literal text from the format string was printed exactly as it was entered.

Carriage Control Escape Sequences Wait a minute. What is that `#` doing at the end of the output? Well, that's our shell prompt. You see, we didn't direct RPM to move to a new line after producing the output, so the shell prompt ended up being tacked to the end of our output.

Is there a way to fix that? Yes, there is. We need to use an escape sequence. An escape sequence is a sequence of characters that starts with a backslash (\). Escape sequences add carriage control information to a format string. The following escape sequences can be used:

- `\a` — Produces a bell or similar alert.
- `\b` — Backspaces one character.
- `\f` — Outputs a form-feed character.
- `\n` — Outputs a newline character sequence.
- `\r` — Outputs a carriage return character.
- `\t` — Causes a horizontal tab.
- `\v` — Causes a vertical tab.
- `\\` — Displays a backslash character.

Based on this list, it seems that a `\n` escape sequence at the end of the format string will put our shell prompt on the next line:

```
# rpm -q -queryformat 'This is a test!\n' rpm
This is a test!
#
```

Much better...

Tags The most important parts of a format string are the tags. Each tag specifies what information is to be displayed and can optionally include field-width, as well as justification and data formatting instructions.[5] But for now, let's look at the basic tag. In fact, let's look at three — the tags that print the package name, version, and release.

Strangely enough, these tags are called `NAME`, `VERSION`, and `RELEASE`. In order to be used in a format string, the tag names must be enclosed in curly braces and preceded by a percent sign. Let's give it a try:

```
# rpm -q --queryformat '%{NAME}%{VERSION}%{RELEASE}\n' rpm
rpm2.31
#
```

Let's add a dash between the tags and see if that makes the output a little easier to read:

```
# rpm -q --queryformat '%{NAME}-%{VERSION}-%{RELEASE}\n' rpm
rpm-2.3-1
#
```

Now our format string outputs standard package labels.

Field Width and Justification Sometimes it's desirable to allocate fields of a particular size for a tag. This is done by putting the desired field width between the tag's leading percent sign, and the opening curly brace. Using our package-label-producing format string, let's allocate a 20-character field for the version:

```
# rpm -q --queryformat '%{NAME}-%20{VERSION}-%{RELEASE}\n' rpm
rpm-                 2.3-1
#
```

The result is a field of 20 characters: 17 spaces, followed by the three characters that make up the version.

[5] RPM uses `printf` to do `--queryformat` formatting. Therefore, you can use any of the `printf` format modifiers discussed in the `printf(3)` man page.

In this case, the version field is right justified; that is, the data is printed at the far right of the output field. We can left justify the field by preceding the field width specification with a dash:

```
# rpm -q --queryformat '%{NAME}-%-20{VERSION}-%{RELEASE}\n' rpm
rpm-2.3                 -1
#
```

Now the version is printed at the far left of the output field. You might be wondering what would happen if the field width specification didn't leave enough room for the data being printed. The field width specification can be considered the *minimum* width the field will take. If the data being printed is wider, the field will expand to accommodate the data.

Modifiers — Making Data More Readable While RPM does its best to appropriately display the data from a `--queryformat`, there are times when you'll need to lend a helping hand. Here's an example. Say we want to display the name of each installed package, followed by the time the package was installed. Looking through the available tags, we see `INSTALLTIME`. Great! Looks like this will be simple:

```
# rpm -qa --queryformat '%{NAME} was installed on %{INSTALLTIME}\n'
setup was installed on 845414601
pamconfig was installed on 845414602
filesystem was installed on 845414607
...
rpm was installed on 851659311
pgp was installed on 846027549
#
```

Well, that's a *lot* of output, but not very useful. What *are* those numbers? RPM didn't lie – they're the time the packages were installed. The problem is, the times are being displayed in their numeric form used internally by the operating system, and humans like to see the day, month, year, and so on.

Fortunately, there's a modifier for just this situation. The name of the modifier is `:date`, and it follows the tag name. Let's try our example again, this time using `:date`:

```
# rpm -qa --queryformat '%{NAME} was installed on %{INSTALLTIME:date}\n'
setup was installed on Tue Oct 15 17:23:21 1996
pamconfig was installed on Tue Oct 15 17:23:22 1996
filesystem was installed on Tue Oct 15 17:23:27 1996
...
rpm was installed on Thu Dec 26 23:01:51 1996
pgp was installed on Tue Oct 22 19:39:09 1996
#
```

That sure is a lot easier to understand, isn't it?

Here's a list of the available modifiers:

- The `:date` modifier displays dates in human-readable form. It transforms `846027549` into `Tue Oct 22 19:39:09 1996`.
- The `:perms` modifier displays file permissions in an easy-to-read format. It changes `-32275` to `-rwxr-xr-x-`.
- The `:depflags` modifier displays the version comparison flags used in dependency processing, in human-readable form. It turns `12` into `>=`.
- The `:fflags` modifier displays a `c` if the file has been marked as being a configuration file, a `d` if the file has been marked as being a documentation file, and blank otherwise. Thus, `2` becomes `d`.

Array Iterators Until now, we've been using tags that represent single data items. There is, for example, only one package name or installation date for each package. However, there are other tags that can represent many different pieces of data. One such tag is `FILENAMES`, which can be used to display the names of every file contained in a package.

Let's put together a format string that will display the package name, followed by the name of every file that package contains. We'll try it on the `adduser` package first, since it contains only one file:

```
# rpm -q --queryformat '%{NAME}: %{FILENAMES}\n' adduser
adduser: /usr/sbin/adduser
#
```

Hey, not bad — got it on the first try. Now let's try it on a package with more than one file:

```
# rpm -q --queryformat '%{NAME}: %{FILENAMES}\n' etcskel
etcskel: (array)
#
```

Hmmm. What went wrong? It worked before... Well, it worked before because the `adduser` package contained only one file. The `FILENAMES` tag points to an array of names, so when there is more than one file in a package, there's a problem.

But there is a solution. It's called an *iterator*. An iterator can step through each entry in an array, producing output as it goes. Iterators are created when square braces enclose one or more tags and literal text. Since we want to iterate through the `FILENAMES` array, let's enclose that tag in the iterator:

```
# rpm -q --queryformat '%{NAME}: [%{FILENAMES}]\n' etcskel
```

```
etcskel:  /etc/skel/etc/skel/.Xclients/etc/skel/.Xdefaults/etc/skel/.ba
#
```

There was more output — it went right off the screen in one long line. The problem? We didn't include a newline escape sequence inside the iterator. Let's try it again:

```
# rpm -q --queryformat '%{NAME}:  [%{FILENAMES}\n]' etcskel
etcskel:  /etc/skel
/etc/skel/.Xclients
/etc/skel/.Xdefaults
/etc/skel/.bash_logout
/etc/skel/.bash_profile
/etc/skel/.bashrc
/etc/skel/.xsession
#
```

That's more like it. If we wanted, we could put another file-related tag inside the iterator. If we included the FILESIZES tag, we'd be able to see the name of each file, as well as how big it was:

```
# rpm -q --queryformat '%{NAME}:  [%{FILENAMES} (%{FILESIZES} bytes)\n]'
> etcskel
etcskel:  /etc/skel (1024 bytes)
/etc/skel/.Xclients (551 bytes)
/etc/skel/.Xdefaults (3785 bytes)
/etc/skel/.bash_logout (24 bytes)
/etc/skel/.bash_profile (220 bytes)
/etc/skel/.bashrc (124 bytes)
/etc/skel/.xsession (9 bytes)
#
```

That's pretty nice. But it would be even nicer if the package name appeared on each line, along with the filename and size. Maybe if we put the NAME tag inside the iterator:

```
# rpm -q --queryformat '[%{NAME}:  %{FILENAMES} (%{FILESIZES} bytes)\n]'
> etcskel
etcskel:  /etc/skel(parallel array size mismatch)#
```

The error message says it all. The FILENAMES and FILESIZES arrays are the same size. The NAME tag isn't even an array. Of course the sizes don't match!

Iterating Single-Entry Tags If a tag only has one piece of data, it's possible to put it in an iterator and have its one piece of data displayed with every iteration. This is done by preceding the tag name with an equal sign. Let's try it out on our current example:

```
# rpm -q --queryformat '[%{=NAME}: %{FILENAMES} (%{FILESIZES} bytes)\n]'
> etcskel
etcskel:  /etc/skel (1024 bytes)
etcskel:  /etc/skel/.Xclients (551 bytes)
etcskel:  /etc/skel/.Xdefaults (3785 bytes)
etcskel:  /etc/skel/.bash_logout (24 bytes)
etcskel:  /etc/skel/.bash_profile (220 bytes)
etcskel:  /etc/skel/.bashrc (124 bytes)
etcskel:  /etc/skel/.xsession (9 bytes)
#
```

That's about all there is to format strings. Now, if RPM's standard output doesn't give you what you want, you have no reason to complain. Just `--queryformat` it!

In Case You Were Wondering... What's that? You say you don't know what tags are available? You can use RPM's `--querytags` option. When used as the only option (ie, `rpm --querytags`), it produces a list of available tags. It should be noted that RPM displays the complete tag name. For instance, `RPMTAG_ARCH` is the complete name, yet you'll only need to use `ARCH` in your format string. Here's a partial example of the `--querytags` option in action:

```
# rpm --querytags
RPMTAG_NAME
RPMTAG_VERSION
RPMTAG_RELEASE
...
RPMTAG_VERIFYSCRIPT
#
```

Be forewarned: the full list is quite lengthy. At the time this book was written, there were over 70 tags! You'll notice that each tag is printed in uppercase, and is preceded with `RPMTAG_`. If we were to use that last tag, `RPMTAG_VERIFYSCRIPT`, in a format string, it could be specified in any of the following ways:

```
%{RPMTAG_VERIFYSCRIPT}

%{RPMTAG_VerifyScript}

%{RPMTAG_VeRiFyScRiPt}

%{VERIFYSCRIPT}
```

```
%{VerifyScript}

%{VeRiFyScRiPt}
```

The only hard-and-fast rule regarding tags is that if you include the `RPMTAG_` prefix, it *must* be all uppercase. The fourth example above shows the traditional way of specifying a tag — prefix omitted, all uppercase. The choice, however, is yours.

One other thing to keep in mind is that not every package will have every type of tagged information available. In cases where the requested information is not available, RPM will display `(none)` or `(unknown)`. There are also a few tags that, for one reason or another, will not produce useful output when using in a format string. For a comprehensive list of queryformat tags, please see Appendix D on page 385.

5.2.3 Getting a *lot* more information with `-vv`

Sometimes it's necessary to have even *more* information than we can get with `-v`. By adding another `v`, we can start to see more of RPM's inner workings:

```
# rpm -qvv rpm
D: opening database in //var/lib/rpm/
D: querying record number 2341208
rpm-2.3-1
#
```

The lines starting with `D:` have been added by using `-vv`. We can see where the RPM database is located and what record number contains information on the `rpm-2.3-1` package. Following that is the usual output.

In the vast majority of cases, it will not be necessary to use `-vv`. It is normally used by software engineers working on RPM itself, and the output can change without notice. However, it's a handy way to gain insights into RPM's inner workings.

5.2.4 `--root` *<path>*: Use *<path>* As An Alternate Root

Adding `--root` *<path>* to a query command forces RPM to assume that the directory specified by *<path>* is actually the "root" directory. In addition, RPM expects its database to reside in the directory specified by the `dbpath` `rpmrc` file entry, relative to *<path>*.[6]

Normally this option is only used during an initial system install, or when a system has been booted off a "rescue disk", and some packages need to be re-installed in order to restore normal operation.

[6] For more information on `rpmrc` file entries, see Appendix B.

5.2.5 --rcfile *<rcfile>*: Use *<rcfile>* As An Alternate rpmrc File

The `--rcfile` option is used to specify a file containing default settings for RPM. Normally, this option is not needed. By default, RPM uses `/etc/rpmrc` and a file named `.rpmrc`, located in your login directory.

This option would be used if there was a need to switch between several sets of RPM options. Software developer and package builders will be the people using `--rcfile`. For more information on `rpmrc` files, see Appendix B.

5.2.6 --dbpath *<path>*: Use *<path>* To Find RPM Database

In order for RPM to do its handiwork, it needs access to an RPM database. Normally, this database exists in the directory specified by the `rpmrc` file entry, `dbpath`. By default, `dbpath` is set to `/var/lib/rpm`.

Although the `dbpath` entry can be modified in the appropriate `rpmrc` file, the `--dbpath` option is probably a better choice when the database path needs to be changed temporarily. An example of a time the `--dbpath` option would come in handy is when it's necessary to examine an RPM database copied from another system. Granted, it's not a common occurrence, but it's difficult to handle any other way.

5.3 A Few Handy Queries

Below are some examples of situations you might find yourself in, and ways you can use RPM to get the information you need. Keep in mind that these are just examples. Don't be afraid to experiment!

5.3.1 Finding Config Files Based on a Program Name

You're setting up a new system, and you'd like to implement some system-wide aliases for people using the Bourne Again SHell, `bash`. The problem is you just can't remember the name of the system-wide initialization file used by `bash`, or where it resides:

```
# rpm -qcf /bin/bash
/etc/bashrc
#
```

Rather than spending time trying to hunt down the file, RPM finds it for you in seconds.

5.3.2 Learning More About an Uninstalled Package

Practically any option can be combined with `-qp` to extract information from a .rpm file. Let's say you have an unknown .rpm file, and you'd like to know a bit more before installing it:

```
# rpm -qpil foo.bar
Name        : rpm                   Distribution: Red Hat Linux Vanderbilt
Version     : 2.3                         Vendor: Red Hat Software
Release     : 1                       Build Date: Tue Dec 24 09:07:59 1996
Install date: (none)                  Build Host: porky.redhat.com
Group       : Utilities/System        Source RPM: rpm-2.3-1.src.rpm
Size        : 631157
Summary     : Red Hat Package Manager
Description :
RPM is a powerful package manager, which can be used to build, install,
query, verify, update, and uninstall individual software packages. A
package consists of an archive of files, and package information,
including name, version, and description.
/bin/rpm
/usr/bin/find-provides
/usr/bin/find-requires
/usr/bin/gendiff
/usr/bin/rpm2cpio
/usr/doc/rpm-2.3-1
...
/usr/src/redhat/SOURCES
/usr/src/redhat/SPECS
/usr/src/redhat/SRPMS
#
```

By displaying the package information, we know that we have a package file containing RPM version 2.3. We can then peruse the file list, and see exactly what it would install before installing it.

5.3.3 Finding Documentation for a Specific Package

Picking on `bash` some more, you realize that your knowledge of the software is lacking. You'd like to see when it was installed on your system, and what documentation is available for it:

```
# rpm -qid bash
Name        :bash                   Distribution: Red Hat Linux (Picasso)
Version     :1.14.6                       Vendor: Red Hat Software
Release     :2                        Build Date: Sun Feb 25 13:59:26 1996
Install date:Mon May 13 12:47:22 1996 Build Host: porky.redhat.com
Group       :Shells                   Source RPM: bash-1.14.6-2.src.rpm
Size        :486557
Description :GNU Bourne Again Shell (bash)
/usr/doc/bash-1.14.6-2
```

```
/usr/doc/bash-1.14.6-2/NEWS
/usr/doc/bash-1.14.6-2/README
/usr/doc/bash-1.14.6-2/RELEASE
/usr/info/bash.info.gz
/usr/man/man1/bash.1
#
```

You never realized that there could be so much documentation for a shell!

5.3.4 Finding Similar Packages

Looking at bash's information, we see that it belongs to the group "Shells". You're not sure what other shell packages are installed on your system. If you can find other packages in the "Shells" group, you'll have found the other installed shells:

```
# rpm -qa --queryformat '%10{NAME} %20{GROUP}\n' | grep -i shells
       ash                Shells
      bash                Shells
       csh                Shells
        mc                Shells
      tcsh                Shells
```

Now you can query each of these packages, and learn more about them, too.[7]

5.3.5 Finding Recently Installed Packages, Part I

You remember installing a new package a few days ago. All you know for certain is that the package installed a new command in the /bin directory. Let's try to find the package:

```
# find /bin -type f -mtime -14 | rpm -qF
rpm-2.3-1
#
```

Looks like RPM version 2.3 was installed sometime in the last two weeks.

5.3.6 Finding Recently Installed Packages, Part II

Another way to see which packages were recently installed is to use the --queryformat option:

```
# rpm -qa --queryformat\
```

[7]Did you see this example and say to yourself, "Hey, they could've used the -g option to query for that group directly"? If you did, you've been paying attention. This is a more general way of searching the RPM database for information: we just happened to search by group in this example.

```
> '%{installtime} %{name}-%{version}-%{release} %{installtime:date}\n'\
> | sort -nr +1 | sed -e 's/^[^ ]* //'
rpm-devel-2.3-1 Thu Dec 26 23:02:05 1996
rpm-2.3-1 Thu Dec 26 23:01:51 1996
pgp-2.6.3usa-2 Tue Oct 22 19:39:09 1996
...
pamconfig-0.50-5 Tue Oct 15 17:23:22 1996
setup-1.5-1 Tue Oct 15 17:23:21 1996
#
```

By having RPM include the installation time in numeric form, it was simple to sort the packages and then use sed to remove the user-unfriendly numeric time.

5.3.7 Finding the Largest Installed Packages

Let's say that you're running low on disk space, and you'd like to see what packages you have installed, along with the amount of space each package takes up. You'd also like to see the largest packages first, so you can get back as much disk space as possible:

```
# rpm -qa --queryformat\
> '%{name}-%{version}-%{release} %{size}\n'\
>  | sort -nr +1
kernel-source-2.0.18-5 20608472
tetex-0.3.4-3 19757371
emacs-el-19.34-1 12259914
...  rootfiles-1.3-1 3494
mkinitrd-1.0-1 1898
redhat-release-4.0-1 22
#
```

If you don't build custom kernels, or use TeX, it's easy to see how much space could be reclaimed by removing those packages.

Chapter 6

Using RPM to Verify Installed Packages

`rpm -V` (or `--verify, or -y`) *options*		
Package Selection Options		Page
`pkg1 ... pkgN`	Verify named package(s)	84
`-p` *<file>*	Verify against package file *<file>*	86
`-f` *<file>*	Verify package owning *<file>*	85
`-a`	Verify all installed packages	85
`-g` *<group>*	Verify packages belonging to group *<group>*	86
Verify—specific Options		Page
`--noscripts`	Do not execute verification script	87
`--nodeps`	Do not verify dependencies	86
`--nofiles`	Do not verify file attributes	88
General Options		Page
`-v`	Display additional information	88
`-vv`	Display debugging information	89
`--root` *<path>*	Set alternate root to *<path>*	90
`--rcfile` *<rcfile>*	Set alternate rpmrc file to *<rcfile>*	90
`--dbpath` *<path>*	Use *<path>* to find the RPM database	90

6.1 `rpm -V` — What Does it Do?

From time to time, it's necessary to make sure that everything on your system is "OK". Are you sure the packages you've installed are still configured properly? Have there been any changes made that you don't know about? Did you mistakenly start a recursive delete in `/usr` and now have to assess the damage?

RPM can help. It can alert you to changes made to any of the files installed by RPM. Also, if a package requires capabilities provided by another package, it can make sure the other package is installed, too.

The command `rpm -V` (The options `-y` and `--verify` are equivalent) verifies an installed package. Before we see how this is done, let's take a step back and look at the big picture.

Every time a package is installed, upgraded, or erased, the changes are logged in RPM's database. It's necessary for RPM to keep track of this information; otherwise it wouldn't be able to perform these operations correctly. You can think of the RPM database (and the disk space it consumes) as being the "price of admission" for the easy package management that RPM provides.[1]

The RPM database reflects the configuration of the system on which it resides. When RPM accesses the database to see how files should be manipulated during an install, upgrade, or erase, it is using the database as a mirror of the system's configuration.

However, we can also use the system configuration as a mirror of the RPM database. What does this "backward" view give us? What purpose would be served?

The purpose would be to see if the system configuration accurately reflects the contents of the RPM database. If the system configuration *doesn't* match the database, then we can reach one of two conclusions:

1. The RPM database has become corrupt. The system configuration is unchanged.
2. The RPM database is intact. The system configuration has changed.

While it would be foolish to state that an RPM database has *never* become corrupt, it is a sufficiently rare occurrence that the second conclusion is much more likely. So RPM gives us a powerful verification tool, essentially for free.

6.1.1 What Does it Verify?

It would be handy if RPM did nothing more than verify that every file installed by a package actually exists on your system. In reality, RPM does much more. It makes sure that if a package depends on other packages to provide certain capabilities, the necessary packages are, in fact, installed. If the package builder created one, RPM will also run a special verification script that can verify aspects of the package's installation that RPM cannot.

Finally, every file installed by RPM is examined. No less than *nine* different attributes of each file can be checked. Here is the list of attributes:

- Owner
- Group
- Mode
- MD5 Checksum

[1] Actually, the price is fairly low. For a completely RPM-based Linux distribution, it would be unusual to have a database over 5MB in size.

- Size
- Major Number
- Minor Number
- Symbolic Link String
- Modification Time

Let's take a look at each of these attributes and why they are good things to check:

File Ownership

Most operating systems today keep track of each file's creator. This is done primarily for resource accounting. Linux and UNIX also use file ownership to help determine access rights to the file. In addition, some files, when executed by a user, can temporarily change the user's ID, normally to a more privileged ID. Therefore, any change of file ownership may have far reaching effects on data security and system availability.

File Group

In a similar manner to file ownership, a "group" specification is attached to each file. Primarily used for determining access rights, a file's group specification can also become a user's group ID, should that user execute the file's contents. Therefore, any changes in a file's group specification are important, and should be monitored.

File Mode

Encompassing the file's "permissions", the mode is a set of bits that specifies permitted access for the file's owner, group members, and everyone else. Even more important are two additional bits that determine whether a user's group or user ID should be changed if they execute the program contained in the file. Since these little bombshells can let any user become `root` for the duration of the program, it pays to be extra careful with a file's permissions.

MD5 Checksum

The MD5 checksum of a file is simply a 128-bit number that is mathematically derived from the contents of the file. The MD5 algorithm was designed by Ron Rivest, the "R" in the popular RSA public-key encryption algorithm. The "MD" in "MD5" stands for *Message Digest*, which is a pretty accurate description of what it does.

Unlike literary digests, an MD5 checksum conveys no information about the contents of the original file. However, it possesses one unique trait:

- *Any* change to the file, no matter how small, results in a change to the MD5 checksum.[2]

RPM creates MD5 checksums of all files it manipulates, and stores them in its database. For all intents and purposes, if one of these files is changed, the MD5 checksum will change, and RPM will detect it.

File Size

As if the use of MD5 isn't enough, RPM also keeps track of file sizes. A difference of even one byte more or less will not go unnoticed.

Major Number

Device character and block files possess a major number. The major number is used to communicate information to the device driver associated with the special file. For instance, under Linux the special files for SCSI disk drives should have a major number of 8, while the major number for an IDE disk drive's special file would be 3. As you can imagine, any change to a file's major number can have disastrous effects, and is tracked by RPM.

Minor Number

A file's minor number is similar in concept to the major number, but conveys different information to the device driver. In the case of disk drives, this information can consist of a unit identifier. Should the minor number change, RPM will detect it.

Symbolic Link

If the file in question is really a symbolic link, the text string containing the name of the linked-to file is checked.

Modification Time

Most operating systems keep track of the date and time that a file was last modified. RPM uses this to its advantage by keeping modification times in its database.

6.2 When Verification Fails - `rpm -V` Output

When verifying a package, RPM produces output *only* if there is a verification failure. When a file fails verification, the format of the output is a bit cryptic, but it packs all the information you need into one line per file. Here is the format:

[2]From a strictly theoretical standpoint, this is not entirely true. Using the lingo of cryptologists, it is believed to be "computationally infeasible" to find two messages that produce the same MD5 checksum.

`SM5DLUGT c` *`file`*

Where:

- `S` is the file size.
- `M` is the file's mode.
- `5` is the MD5 checksum of the file.
- `D` is the file's major and minor numbers.
- `L` is the file's symbolic link contents.
- `U` is owner of the file.
- `G` is the file's group.
- `T` is the modification time of the file.
- `c` appears only if the file is a configuration file. This is handy for quickly identifying config files, as they are very likely to change, and therefore, very *unlikely* to verify successfully.
- *`file`* is the file that failed verification. The complete path is listed to make it easy to find.

It's unlikely that *every* file attribute will fail to verify, so each of the eight attribute flags will only appear if there is a problem. Otherwise, a "`.`" will be printed in that flag's place. Let's look at an example or two:

```
.M5....T   /usr/X11R6/lib/X11/fonts/misc/fonts.dir
```

In this case, the mode, MD5 checksum, and modification time for the specified file have failed to verify. The file is not a config file (Note the absence of a "`c`" between the attribute list and the filename).

```
S.5....T c /etc/passwd
```

Here, the size, checksum, and modification time of the system password file have all changed. The "`c`" indicates that this is a config file.

```
missing    /var/spool/at/spool
```

This last example illustrates what RPM does when a file, that should be there, is missing entirely.

6.2.1 Other Verification Failure Messages

When `rpm -V` finds other problems, the output is a bit easier to understand:

```
# rpm -V blather
Unsatisfied dependencies for blather-7.9-1:  bother >= 3.1
#
```

It's pretty easy to see that the `blather` package requires at least version 3.1 of the `bother` package.

The output from a package's verification script is a bit harder to categorize, as the script's contents, as well as its messages, are entirely up to the package builder.

6.3 Selecting What to Verify, and How

There are several ways to verify packages installed on your system. If you've taken a look at RPM's query command, you'll find that many of them are similar. Let's start with the simplest method of specifying packages — the package label.

The Package Label — Verify an Installed Package Against the RPM Database

You can simply follow the `rpm -V` command with all or part of a package label. As with every other RPM command that accepts package labels, you'll need to carefully specify each part of the label you include. Keep in mind that package names are case-sensitive, so `rpm -V PackageName` and `rpm -V packagename` are *not* the same. Let's verify the `initscripts` package:

```
# rpm -V initscripts
#
```

While it looks like RPM didn't do anything, the following steps were performed:

- For every file in the package, RPM checked the nine file attributes that were discussed above.
- If the package was built with dependencies, the RPM database was searched to ensure the packages that satisfy those dependencies were installed.
- If the package was built with a verification script, that script was executed.

In our example, each of these steps was performed without error — the package verified successfully. Remember, with `rpm -V` you'll only see output if a package fails to verify.

-a — Verify All Installed Packages Against the RPM Database

If you add `-a` to `rpm -V`, you can easily verify every installed package on your system. It might take a while, but when it's done, you'll know exactly what's been changed on your system:

```
# rpm -Va
.M5....T   /usr/X11R6/lib/X11/fonts/misc/fonts.dir
missing    /var/spool/at/.lockfile
missing    /var/spool/at/spool
S.5....T   /usr/lib/rhs/glint/icon.pyc
..5....T c /etc/inittab
..5.....   /usr/bin/loadkeys
#
```

Don't be too surprised if `rpm -Va` turns up a surprising number of files that failed verification. RPM's verification process is *very* strict! In many cases, the changes flagged don't indicate problems — they are only an indication of your system's configuration being different than what the builders of the installed packages had on *their* system. Also, some attributes change during normal system operation. However, it would be wise to check into each verification failure, just to make sure.

-f *<file>* — Verify the Package Owning *<file>* Against the RPM Database

Imagine this: you're hard at work when a program you've used a million times before suddenly stops working. What do you do? Well, before using RPM, you probably tried to find other files associated with that program and see if they had changed recently.

Now you can let RPM do at least part of that sleuthing for you. Simply direct RPM to verify the package owning the ailing program:

```
% rpm -Vf /sbin/cardmgr
S.5....T c /etc/sysconfig/pcmcia
```

Hmmmm. Looks like a config file was recently changed.

This isn't to say that using RPM to verify a package will always get you out of trouble, but it's such a quick step it should be one of the first things you try. Here's an example of `rpm -Vf` not working out as well:

```
% rpm -Vf /etc/blunder
file /etc/blunder is not owned by any package
```

(Note that the issue surrounding RPM and symbolic links mentioned in "A Tricky Detail" (page 55) also applies to `rpm -Vf`. Watch those symlinks!)

-p *<file>* — Verify Against a Specific Package File

Unlike the previous options to `rpm -V`, each of which verified one or more packages against RPM's database, the `-p` option performs the same verification, but against a package file. Why on earth would you want to do this when the RPM database is sitting there just waiting to be used?

Well, what if you didn't *have* an RPM database? While it isn't a common occurrence, power failures, hardware problems, and inadvertent deletions (along with non-existent backups) can leave your system "sans database". Then your system hiccups — what do you do now?

This is where a CD full of package files can be worth its weight in gold. Simply mount the CD and verify to your heart's content:

```
# rpm -Vp /mnt/cdrom/RedHat/RPMS/i386/adduser-1.1-1.i386.rpm
#
```

Whatever else might be wrong with this system, at least we can add new users. But what if you have *many* packages to verify? It would be a very slow process doing it one package at a time. That's where the next option comes in handy...

-g *<group>* — Verify Packages Belonging To *<group>*

When a package is built, the package builder must classify the package, grouping it with other packages that perform similar functions. RPM gives you the ability to verify installed packages based on their groups. For example, there is a group known as `Shells`. This group consists of packages that contain, strangely enough, shells. Let's verify the proper installation of every shell-related package on the system:

```
# rpm -Vg Shells
missing /etc/bashrc
#
```

One thing to keep in mind is that group specifications are case-sensitive. Issuing the command `rpm -Vg shells` wouldn't verify many packages:

```
# rpm -Vg shells
group shells does not contain any packages
#
```

--nodeps: Do Not Check Dependencies Before Erasing Package

When the `--nodeps` option is added to a verify command, RPM will bypass its dependency verification processing. In this example, we've added the `-vv` option to so we can watch RPM at work:

```
# rpm -Vvv rpm
D: opening database in //var/lib/rpm/
D: verifying record number 2341208
D: dependencies:  looking for libz.so.1
D: dependencies:  looking for libdb.so.2
D: dependencies:  looking for libc.so.5
#
```

As we can see, there are three different capabilities that the `rpm` package provides:

- `libz.so.1`
- `libdb.so.2`
- `libc.so.5`

If we add the `--nodeps` option, the dependency verification of the three capabilities is no longer performed:

```
# rpm -Vvv --nodeps rpm
D: opening database in //var/lib/rpm/
D: verifying record number 2341208
#
```

The line `D: verifying record number 2341208` indicates that RPM's normal file-based verification proceeded normally.

`--noscripts`: Do Not Execute Verification Script

Adding the `--noscripts` option to a verify command prevents execution of the verification scripts of each package being verified. In the following example, the package verification script is executed:

```
# rpm -Vvv bother
D: opening database in //var/lib/rpm/
D: verifying record number 616728
D: verify script found - running from file /var/tmp/rpm-321.vscript
+ PATH=/sbin:/bin:/usr/sbin:/usr/bin:/usr/X11R6/bin
+ export PATH
+ echo This is the bother 3.5 verification script
This is the bother 3.5 verification script
#
```

While the actual script is not very interesting, it did execute when the package was being verified. In the next example, we'll use the `--noscripts` option to prevent its execution:

```
# rpm -Vvv --noscripts bother
D: opening database in //var/lib/rpm/
D: verifying record number 616728
#
```

As expected, the output is identical to the prior example — minus the lines dealing with the verification script, of course.

`--nofiles`: Do Not Verify File Attributes

The `--nofiles` option disables RPM's file-related verification processing. When this option is used, only the verification script and dependency verification processing are performed. In this example, the package has a file-related verification problem:

```
# rpm -Vvv bash
D: opening database in //var/lib/rpm/
D: verifying record number 279448
D: dependencies:  looking for libc.so.5
D: dependencies:  looking for libtermcap.so.2
missing /etc/bashrc
#
```

When the `--nofiles` option is added, the missing file doesn't cause a message any more:

```
# rpm -Vvv --nofiles bash
D: opening database in //var/lib/rpm/
D: verifying record number 279448
D: dependencies:  looking for libc.so.5
D: dependencies:  looking for libtermcap.so.2
#
```

This is not to say that the missing file problem is solved, just that no file verification was performed.

`-v` — Display Additional Information

Although RPM won't report an error with the command syntax if you include the `-v` option, you won't see much in the way of additional output:

```
# rpm -Vv bash
```

```
#
```

Even if there are verification errors, adding `-v` won't change the output:

```
# rpm -Vv apmd
S.5....T /etc/rc.d/init.d/apm
S.5....T /usr/X11R6/bin/xapm
#
```

The only time that the `-v` option *will* produce output is when the package being verified has a verification script. Any normal output from the script won't be displayed by RPM, when run without `-v`:[3]

```
# rpm -V bother
#
```

But when `-v` is added, the script's non-error-related output is displayed:

```
# rpm -Vv bother
This is the bother 3.5 verification script
#
```

If you're looking for more insight into RPM's inner workings, you'll have to try the next option:

`-vv` — Display Debugging Information

Sometimes it's necessary to have even *more* information than we can get with `-v`. By adding another `v`, that's just what we'll get:

```
# rpm -Vvv rpm
D: opening database in //var/lib/rpm/
D: verifying record number 2341208
D: dependencies:  looking for libz.so.1
D: dependencies:  looking for libdb.so.2
D: dependencies:  looking for libc.so.5
#
```

The lines starting with `D:` have been added by using `-vv`. We can see where the RPM database is located and what record number contains information on the `rpm-2.3-1` package. Following that is the list of dependencies that the `rpm` package requires.

[3] Failure messages will always be displayed.

In the vast majority of cases, it will not be necessary to use `-vv`. It is normally used by software engineers working on RPM itself, and the output can change without notice. However, it's a handy way to gain insights into RPM.

`--dbpath` *<path>*: Use *<path>* To Find RPM Database

In order for RPM to do its handiwork, it needs access to an RPM database. Normally, this database exists in the directory specified by the `rpmrc` file entry, `dbpath`. By default, `dbpath` is set to `/var/lib/rpm`.

Although the `dbpath` entry can be modified in the appropriate `rpmrc` file, the `--dbpath` option is probably a better choice when the database path needs to be changed temporarily. An example of a time the `--dbpath` option would come in handy is when it's necessary to examine an RPM database copied from another system. Granted, it's not a common occurrence, but it's difficult to handle any other way.

`--root` *<path>*: Set Alternate Root to *<path>*

Adding `--root` *<path>* to a verify command forces RPM to assume that the directory specified by *<path>* is actually the "root" directory. In addition, RPM expects its database to reside in the directory specified by the `dbpath` `rpmrc` file entry, relative to *<path>*.[4]

Normally this option is only used during an initial system install, or when a system has been booted off a "rescue disk", and some packages need to be re-installed in order to restore normal operation.

`--rcfile` *<rcfile>*: Set Alternate `rpmrc` file to *<rcfile>*

The `--rcfile` option is used to specify a file containing default settings for RPM. Normally, this option is not needed. By default, RPM uses `/etc/rpmrc` and a file named `.rpmrc`, located in your login directory.

This option would be used if there was a need to switch between several sets of RPM options. Software developer and package builders will be the people using `--rcfile`. For more information on `rpmrc` files, see Appendix B.

6.4 We've Lied to You...

Not really; we just omitted a few details until you've had a chance to see `rpm -V` in action. Here are the details:

RPM Controls What Gets Verified Depending on the type of file being verified, RPM will not verify every possible attribute. Here is a table showing the attributes checked for each of the different file types:

[4] For more information on `rpmrc` file entries, see Appendix B.

File Type	File Size	Mode	MD5 Checksum	Major Number	Minor Number	Symlink String	Owner	Group	Modification Time
Directory File		X					X	X	
Symbolic Links		X				X	X	X	
FIFO		X					X	X	
Devices		X		X	X		X	X	
Regular Files	X	X	X				X	X	X

The Package Builder Can Also Control What Gets Verified When a package builder creates a new package, they can control what attributes are to be verified on a file-by-file basis. The reasons for excluding specific attributes from verification can be quite involved, but here's an example just to give you the flavor:

When a person logs into a system, there are device files associated with that user's terminal session. In order for the terminal device (called `tty`) to function properly, the owner and group of the device must change to that of the person logging in. Therefore, if RPM were to verify the package that created the `tty` device files, any ttys that were in use at the time would fail to verify. However, by using the `%verify`[5] directive, a package builder can save you from trivial verification failures.

[5] See page 200 for details on `%verify`

Chapter 7

Using RPM to Verify Package Files

rpm -K (or --checksig) *options* file1.rpm ... fileN.rpm		
Parameters		
file1.rpm ... fileN.rpm	One or more RPM package files (URLs OK)	
Checksig—specific Options		Page
--nopgp	Do not verify PGP signatures	99
General Options		Page
-v	Display additional information	96
-vv	Display debugging information	99
--rcfile *<rcfile>*	Set alternate rpmrc file to *<rcfile>*	100

7.1 rpm -K — What Does it Do?

One aspect of RPM is that you can get a package from the Internet, and easily install it. But what do you know about that package file? Is the organization listed as being the "vendor" of the package *really* the organization that built it? Did someone make unauthorized changes to it? Can you trust that, if installed, it won't mail a copy of your password file to a system cracker?

Features built into RPM allow you to make sure that the package file you've just gotten won't cause you problems once it's installed, whether the package was corrupted by line noise when you downloaded it, or something more sinister happened to it.

The command `rpm -K` (The option `--checksig` is equivalent) verifies a package file. Using this command, it is easy to make sure the file has not been changed in any way. `rpm -K` can also be used to make sure that the package was actually built by the organization listed as being the package's vendor. That's all very impressive, but how does it do that? Well, it just needs help from some "Pretty Good" software.

7.1.1 Pretty Good Privacy: RPM's Assistant

The "Pretty Good" software we're referring to is known as "Pretty Good Privacy", or PGP. While all the information on PGP could fill a book (or several), we've provided a quick introduction to help you get started.

If PGP is new to you, a quick glance through Appendix G should get you well on your way to understanding, building, and installing PGP. If, on the other hand, you've got PGP already installed and have sent an encrypted message or two, you're probably more than ready to continue with this chapter.

7.2 Configuring PGP for `rpm -K`

Once PGP is properly built and installed, the actual configuration for RPM is trivial. Here's what needs to be done:

- PGP must be in your path. If PGP's usage message doesn't come up when you enter `pgp` at your shell prompt, you'll need to add PGP's directory to your path.
- PGP must be able to find the public keyring file that you want to use when checking package file signatures. You can use two methods to direct PGP to the public keyring:
  1. Set the `PGPPATH` environment variable to point to the directory containing the public keyring file.
  2. Set the `pgp_path` `rpmrc` file entry to point to the directory containing the public keyring file.[1]

Now we're ready.

7.3 Using `rpm -K`

After all the preliminaries with PGP, it's time to get down to business. First, we need to get the package builder's public key and add it to the public keyring file used by RPM. You'll need to do this once for each package builder whose packages you'll want to check. This is what you'll need to do:

```
# pgp -ka RPM-PGP-KEY ./pubring.pgp
Pretty Good Privacy(tm) 2.6.3a - Public-key encryption for the masses.
(c) 1990-96 Philip Zimmermann, Phil's Pretty Good Software.  1996-03-04
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
```

[1] For more information on `rpmrc` files, `rpmrc` file entries, and how to use them, please see Appendix B on page 363.

```
Current time:  1996/06/01 22:50 GMT

Looking for new keys...
pub 1024/CBA29BF9 1996/02/20 Red Hat Software, Inc.  <redhat@redhat.com>

Checking signatures...


Keyfile contains:
1 new key(s)

One or more of the new keys are not fully certified.
Do you want to certify any of these keys yourself (y/N)? n
```

Here we've added Red Hat Software's public key, since we're going to check some package files produced by them. The file `RPM-PGP-KEY` contains the key. At the end, PGP asks us if we want to certify the new key. We've answered "no" since it isn't necessary to certify keys to verify package files.

Next, we'll verify a package file:

```
# rpm -K rpm-2.3-1.i386.rpm
rpm-2.3-1.i386.rpm:  size pgp md5 OK
#
```

While the output might seem somewhat anti-climactic, we can now be nearly 100% certain this package:

1. was produced by Red Hat Software, Inc.
2. is unchanged from their original copy.

The output from this command shows that there are actually three distinct features of the package file that are checked by the `-K` option:

1. The `size` message indicates that the size of the packaged files has not changed.
2. The `pgp` message indicates that the digital signature contained in the package file is a valid signature of the package file contents, and was produced by the organization that originally signed the package.
3. The `md5` message indicates that a checksum contained in the package file and calculated when the package was built, matches a checksum calculated by RPM during verification. Because the two checksums match, it is unlikely that the package has been modified.

The OK means that each of these tests were successful. If any had failed, the name would have been printed in parentheses. A bit later in the chapter, we'll see what happens when there are verification problems.

7.3.1 -v — Display Additional Information

Adding v to a verification command will produce more interesting output:

```
# rpm -Kv rpm-2.3-1.i386.rpm
rpm-2.3-1.i386.rpm:
Header+Archive size OK: 278686 bytes
Good signature from user "Red Hat Software, Inc.  <redhat@redhat.com>".
Signature made 1996/12/24 18:37 GMT using 1024-bit key, key ID CBA29BF9

WARNING: Because this public key is not certified with a trusted
signature, it is not known with high confidence that this public key
actually belongs to:  "Red Hat Software, Inc.  <redhat@redhat.com>".
MD5 sum OK: 8873682c5e036a307dee87d990e75349
#
```

With a bit of digging, we can see that each of the three tests was performed, and each passed. The reason for that dire-sounding warning is that PGP is meant to operate without a central authority managing key distribution. PGP certifies keys based on *webs of trust*. For example, if an acquaintance of yours creates a public key, you can certify it by attaching your digital signature to it. Then anyone that knows and trusts you can also trust your acquaintance's public key.

In this case, the key came directly from a mass-produced Red Hat Linux CDROM. If someone was trying to masquerade as Red Hat Software, Inc. then they have certainly gone through a lot of trouble to do so. In this case, the lack of a certified public key is not a major problem, given the fact that the CDROM came directly from the Red Hat Software offices.[2]

7.3.2 When the Package is Not Signed

As mentioned earlier, not every package you'll run across is going to be signed. If this is the case, here's what you'll see from RPM:

```
# rpm -K bother-3.5-1.i386.rpm
bother-3.5-1.i386.rpm:  size md5 OK
#
```

[2] Red Hat Software's public key is also available from their website, at http://www.redhat.com/redhat/contact.html. The RPM sources also contain the key, and are available from their FTP site at ftp://ftp.redhat.com/pub/redhat/code/rpm.

Note the lack of a pgp message. The size and md5 messages indicate that the package still has size and checksum information that verified properly. In fact, all recently-produced package files will have these verification measures built in automatically.

If you happen to run across an older unsigned package, you'll know it right away:

```
# rpm -K apmd-2.4-1.i386.rpm
apmd-2.4-1.i386.rpm:  No signature available
#
```

Older package files had only a PGP-based signature; if that was missing, there was nothing left to verify.

7.3.3 When You Are Missing the Correct Public Key

If you happen to forget to add the right public key to RPM's keyring, you'll see the following response:

```
# rpm -K rpm-2.3-1.i386.rpm
rpm-2.3-1.i386.rpm:  size (PGP) md5 OK (MISSING KEYS)
#
```

Here the PGP in parentheses indicates that there's a problem with the signature, and the message at the end of the line (MISSING KEYS) shows what the problem is. Basically, RPM asked PGP to verify the package against a key that PGP didn't have, and PGP complained.

7.3.4 When a Package Just Doesn't Verify

Eventually it's going to happen — you go to verify a package, and it fails. We'll look at an example of a package that fails verification a bit later. Before we do that, let's *make* a package that won't verify, to demonstrate how sensitive RPM's verification is.

First, we made a copy of a signed package, rpm-2.3-1.i386.rpm, to be specific. We called the copy rpm-2.3-1.i386-bogus.rpm. Next, using Emacs (in hexl-mode, for all you Emacs buffs), we changed the first letter of the name of the system that built the original package. The file rpm-2.3-1.i386-bogus.rpm is now truly bogus: it has been changed from the original file.

Although the change was a small one, it still showed up when the package file was queried. Here's a listing from the original package:

```
# rpm -qip rpm-2.3-1.i386.rpm
Name        : rpm                    Distribution: Red Hat Linux Vanderbilt
Version     : 2.3                          Vendor: Red Hat Software
Release     : 1                        Build Date: Tue Dec 24 09:07:59 1996
```

```
Install date: (none)                  Build Host: porky.redhat.com
Group       : Utilities/System        Source RPM: rpm-2.3-1.src.rpm
Size        : 631157
Summary     : Red Hat Package Manager
Description :
RPM is a powerful package manager, which can be used to build, install,
query, verify, update, and uninstall individual software packages. A
package consists of an archive of files, and package information,
including name, version, and description.
#
```

And here's the same listing from the bogus package file:

```
# rpm -qip rpm-2.3-1.i386-bogus.rpm
Name        : rpm                   Distribution: Red Hat Linux Vanderbilt
Version     : 2.3                         Vendor: Red Hat Software
Release     : 1                       Build Date: Tue Dec 24 09:07:59 1996
Install date: (none)                  Build Host: qorky.redhat.com
Group       : Utilities/System        Source RPM: rpm-2.3-1.src.rpm
Size        : 631157
Summary     : Red Hat Package Manager
Description :
RPM is a powerful package manager, which can be used to build, install,
query, verify, update, and uninstall individual software packages. A
package consists of an archive of files, and package information,
including name, version, and description.
#
```

Notice that the build host name changed from `porky.redhat.com` to `qorky.redhat.com`. Using the `cmp` utility to compare the two files, we find that the difference occurs at byte 1201, which changed from "p" (octal 160), to "q" (octal 161):

```
# cmp -cl rpm-2.3-1.i386.rpm rpm-2.3-1.i386-bogus.rpm
  1201 160 p    161 q
#
```

People versed in octal numbers will note that only *one bit* has been changed in the entire file. That's the smallest possible change you can make! Let's see how our bogus friend fares:

```
# rpm -K rpm-2.3-1.i386-bogus.rpm
rpm-2.3-1.i386-bogus.rpm:  size PGP MD5 NOT OK
#
```

Given that the command's output ends with `NOT OK` in big capital letters, it's obvious there's a problem. Since the word `size` was printed in lowercase, the bogus package's size was OK, which makes sense — we only changed the value of one bit without adding or subtracting anything else.

However, the PGP signature, printed in uppercase, didn't verify. Again, this makes sense, too. The package that was signed by Red Hat Software has been changed. The fact that the package's MD5 checksum also failed to verify provides further evidence that the bogus package is just that: bogus.

7.3.5 `--nopgp` — Do Not Verify Any PGP Signatures

Perhaps you want to be able to verify packages but, for one reason or another, you cannot use PGP. Maybe you don't have a trustworthy source of the necessary public keys, or maybe it's illegal to possess encryption (like PGP) software in your country. Is it still possible to verify packages?

Certainly — in fact, we've already done it, in section 7.3.3, on page 97. You lose the ability to verify the package's origins, as well as some level of confidence in the package's integrity, but the size and MD5 checksums still give some measure of assurance as to the package's state.

Of course, when PGP can't be used, the output from a verification always looks like something's wrong:

```
# rpm -K rpm-2.3-1.i386.rpm
rpm-2.3-1.i386.rpm:  size (PGP) md5 OK (MISSING KEYS)
#
```

The `--nopgp` option directs RPM to ignore PGP entirely. If we use the `--nopgp` option on our example above, we find that things look a whole lot better:

```
# rpm -K --nopgp rpm-2.3-1.i386.rpm
rpm-2.3-1.i386.rpm:  size md5 OK
#
```

7.3.6 `-vv` — Display Debugging Information

Nine times out of ten, you'll probably never have to use it, but if you're the curious type, the `-vv` option will give you insights into how RPM verifies packages. Here's an example:

```
# rpm -Kvv rpm-2.3-1.i386.rpm
D: New Header signature
D: magic: 8e ad e8 01
D: got  : 8e ad e8 01
D: Signature size: 236
D: Signature pad : 4
```

```
D: sigsize         : 240
D: Header + Archive: 278686
D: expected size   : 278686
rpm-2.3-1.i386.rpm:
Header+Archive size OK: 278686 bytes
Good signature from user "Red Hat Software, Inc. <redhat@redhat.com>".
Signature made 1996/12/24 18:37 GMT using 1024-bit key, key ID CBA29BF9

WARNING:  Because this public key is not certified with a trusted
signature, it is not known with high confidence that this public key
actually belongs to: "Red Hat Software, Inc. <redhat@redhat.com>".
MD5 sum OK: 8873682c5e036a307dee87d990e75349
#
```

The lines starting with `D:` represent extra output produced by the `-vv` option. This output is normally used by software developers in the course of adding new features to RPM and is subject to change, but there's no law against looking at it.

Briefly, the output shows that RPM has detected a new-style signature block, containing size, MD5 checksum, and PGP signature information. The size of the signature, the size of the package file's header and archive sections, and the expected size of those sections are all displayed.

7.3.7 `--rcfile` <*rcfile*>: Use <*rcfile*> As An Alternate `rpmrc` File

The `--rcfile` option is used to specify a file containing default settings for RPM. Normally, this option is not needed. By default, RPM uses `/etc/rpmrc` and a file named `.rpmrc` located in your login directory.

This option would be used if there was a need to switch between several sets of RPM defaults. Software developers and package builders will normally be the only people using the `--rcfile` option. For more information on `rpmrc` files, see Appendix B.

Chapter 8

Miscellania

As with any other large, complex subject, there are always some leftovers — things that just don't seem to fit in any one category. RPM is no exception. This chapter covers those aspects of RPM that can only be called "miscellania"...

8.1 Other RPM Options

The following options are not normally used on a day to day basis. However, some of them can be quite important when the need arises. One such option is `--rebuilddb`.

8.1.1 `--rebuilddb` — Rebuild RPM database

We all hope the day never comes, and for many of us, it never does. But still, there is a chance that one day, while you're busy using RPM to install or upgrade a package, you'll see this message:

```
free list corrupt (42)- contact rpm-list@redhat.com
```

Once this happens, you'll find there's very little that you can do, RPM-wise. However, before you fire off an e-mail to the RPM mailing list, you might try the `--rebuilddb` option. The format of the command is simple:

`rpm --rebuilddb`

The command produces no output, either. After a few minutes, it completes with nary a peep. Here's an example of `--rebuilddb` being used on an RPM database that wasn't corrupt. First, let's look at the files that comprise the database:

```
# cd /var/lib/rpm
# ls
```

```
total 3534
-rw-r--r--   1 root     root      1351680 Oct 17 10:35 fileindex.rpm
-rw-r--r--   1 root     root        16384 Oct 17 10:35 groupindex.rpm
-rw-r--r--   1 root     root        16384 Oct 17 10:35 nameindex.rpm
-rw-r--r--   1 root     root      2342536 Oct 17 10:35 packages.rpm
-rw-r--r--   1 root     root        16384 Oct 17 10:35 providesindex.rpm
-rw-r--r--   1 root     root        16384 Oct 17 10:35 requiredby.rpm
#
```

Then, we issue the command:

```
# rpm --rebuilddb
#
```

After a few minutes, the command completes, and we take a look at the files again:

```
# ls
total 3531
-rw-r--r--   1 root     root      1351680 Oct 17 20:50 fileindex.rpm
-rw-r--r--   1 root     root        16384 Oct 17 20:50 groupindex.rpm
-rw-r--r--   1 root     root        16384 Oct 17 20:50 nameindex.rpm
-rw-r--r--   1 root     root      2339080 Oct 17 20:50 packages.rpm
-rw-r--r--   1 root     root        16384 Oct 17 20:50 providesindex.rpm
-rw-r--r--   1 root     root        16384 Oct 17 20:50 requiredby.rpm
#
```

You'll note that `packages.rpm` decreased in size. This is due to a side-effect of the `--rebuilddb` option – While it is going through the database, it is getting rid of unused portions of the database. Our example was performed on a newly installed system where only one or two packages had been upgraded, so the reduction in size was small. For a system that has been through a complete upgrade, the difference would be more dramatic.

Does this mean that you should rebuild the database every once in a while? Not really. Since RPM eventually will make use of the holes, there's no major advantage to regular rebuilds. However, when an RPM-based system has undergone a major upgrade, it certainly wouldn't hurt to spend a few minutes using `--rebuilddb` to clean things up.

8.1.2 `--initdb` — Create a New RPM Database

If you are already using RPM, the `--initdb` option is one you'll probably never have to use. The `--initdb` option is used to create a new RPM database. That's why you'll probably not need it if you're already using RPM — you already have an RPM database.

It might seem that the `--initdb` option would be dangerous. After all, won't it trash your current database if you mistakenly use it? Fortunately, the answer is no. If there is an RPM database in place already, it's still perfectly safe to use the option, even though it won't accomplish much. As an example, here's a listing of the files that make up the RPM database on a Red Hat Linux system:

```
# ls /var/lib/rpm
total 3559
-rw-r--r--   1 root     root        16384 Jan  8 22:10 conflictsindex.rpm
-rw-r--r--   1 root     root      1351680 Jan  8 22:10 fileindex.rpm
-rw-r--r--   1 root     root        16384 Jan  8 22:10 groupindex.rpm
-rw-r--r--   1 root     root        16384 Jan  8 22:10 nameindex.rpm
-rw-r--r--   1 root     root      2349640 Jan  8 22:10 packages.rpm
-rw-r--r--   1 root     root        16384 Jan  8 22:10 providesindex.rpm
-rw-r--r--   1 root     root        16384 Jan  8 22:10 requiredby.rpm
#
```

Next, let's use the `--initdb` option, just to see what it does to this database:

```
# rpm --initdb
# ls /var/lib/rpm
total 3559
-rw-r--r--   1 root     root        16384 Jan  8 22:10 conflictsindex.rpm
-rw-r--r--   1 root     root      1351680 Jan  8 22:10 fileindex.rpm
-rw-r--r--   1 root     root        16384 Jan  8 22:10 groupindex.rpm
-rw-r--r--   1 root     root        16384 Jan  8 22:10 nameindex.rpm
-rw-r--r--   1 root     root      2349640 Jan  8 22:10 packages.rpm
-rw-r--r--   1 root     root        16384 Jan  8 22:10 providesindex.rpm
-rw-r--r--   1 root     root        16384 Jan  8 22:10 requiredby.rpm
#
```

Since an RPM database existed already, the `--initdb` option did no harm to it — there was no change to the database files.

The only other option that can be used with `--initdb` is `--dbpath`. This permits the easy creation of a new RPM database in the directory specified with the `--dbpath` option.

8.1.3 `--quiet` — Produce as little output as possible

Adding the `--quiet` option to any RPM command directs RPM to produce as little output as possible. For example, RPM's build command (the subject of the second half of this book) normally produces reams of output; by adding the `--quiet` option, this is all you'll see:

```
# rpm -ba --quiet bother-3.5.spec
Package:  bother
1 block
3 blocks
```

```
#
```

The `--quiet` option can silence even the mighty `-vv` option:

```
# rpm -Uvv --quiet eject-1.2-2.i386.rpm
#
```

8.1.4 `--help` — Display a help message

RPM includes a concise built-in help message for those times when you need a reminder about a particular command. Normally you'll want to use the `--help` option by itself, though you might want to pipe the output through a pager such as `less`, since the output is more than one screen long:

```
# rpm --help|less

RPM version 2.3
Copyright (C) 1995 - Red Hat Software
This may be freely redistributed under the terms of the GNU Public License

usage:
   --help               - print this message
   --version    - print the version of rpm being used
   all modes support the following arguments:
      --rcfile <file>   - use <file> instead of /etc/rpmrc and $HOME/.rpmrc
       -v                     - be a little more verbose
       -vv                    - be incredibly verbose (for debugging)
   -q                   - query mode
      --root <dir>        - use <dir> as the top level directory
      --dbpath <dir>      - use <dir> as the directory for the database
      --queryformat <s>   - use s as the header format (implies -i)
   install, upgrade and query (with -p) allow ftp URL's to be used in place
   of file names as well as the following options:

      --ftpproxy <host>   - hostname or IP of ftp proxy

      --ftpport <port>    - port number of ftp server (or proxy)
```

This is just the first screen of RPM's help command. To see the rest, give the command a try. Practically everything there is to know about RPM is present in the `--help` output. It's a bit too concise to learn RPM from, but it's enough to refresh your memory when the syntax of a particular option escapes you.

8.1.5 --version — Display the current RPM version

If you're not sure what version of RPM is presently installed on your system, the easiest way to find out is to ask RPM itself using the `--version` option:

```
# rpm --version
RPM version 2.3
#
```

8.2 Using rpm2cpio

From time to time, you might find it necessary to extract one or more files from a package file. One way to do this would be to:

- Install the package
- Make a copy of the file(s) you need
- Erase the package

An easier way would be to use `rpm2cpio`.

8.2.1 rpm2cpio — What does it do?

As the name implies, `rpm2cpio` takes an RPM package file and converts it to a `cpio` archive. Because it's written to be used primarily as a filter, there's not much to be specified. `rpm2cpio` takes only only one argument, and even *that's* optional!

The optional argument is the name of the package file to be converted. If there is no filename specified on the command line, `rpm2cpio` will simply read from standard input and convert *that* to a `cpio` archive. Let's give it a try:

```
# rpm2cpio logrotate-1.0-1.i386.rpm
0707020001a86a000081a4000000000000000000000001313118bb000002c200000008000
00003000000000000000000000190000e73eusr/man/man8/logrotate.8." logrotate
- log fi
le rotator
.TH rpm 8 "28 November 1995" "Red Hat Software" "Red Hat Linux"
.SH NAME
```

(We've just shown the first few lines of output.)

What on earth is all that stuff? Remember, `rpm2cpio` is written as a filter. It writes the `cpio` archive contained in the package file to standard output, which, if you've not redirected it somehow, is your screen. Here's a more reasonable example:

```
# rpm2cpio logrotate-1.0-1.i386.rpm > blah.cpio
# file blah.cpio
blah.cpio:  ASCII cpio archive (SVR4 with CRC)
#
```

Here we've directed `rpm2cpio` to convert the `logrotate` package file. We've also redirected `rpm2cpio`'s output to a file called `blah.cpio`. Next, using the `file` command, we find that the resulting file is indeed a true-blue `cpio` archive file. The following command is entirely equivalent to the one above and shows `rpm2cpio`'s ability to read the package file from its standard input:

```
# cat logrotate-1.0-1.i386.rpm | rpm2cpio > blah.cpio
#
```

8.2.2 A more real-world example — Listing the files in a package file

While there's nothing wrong with using `rpm2cpio` to actually create a `cpio` archive file, it's takes a few more steps and uses a bit more disk space than is strictly necessary. A somewhat cleaner approach would be to pipe `rpm2cpio`'s output directly into `cpio`:

```
# rpm2cpio logrotate-1.0-1.i386.rpm | cpio -t
usr/man/man8/logrotate.8
usr/sbin/logrotate
14 blocks
#
```

In this example, we used the `-t` option to direct `cpio` to produce a "table of contents" of the archive created by `rpm2cpio`. This can make it much easier to get the right filename and path when you want to extract a file.

8.2.3 Extracting one or more files from a package file

Continuing the example above, let's extract the man page from the `logrotate` package. In the table of contents, we see that the full path is `usr/man/man8/logrotate.8`. All we need to do is to use the filename and path as shown below:

```
# rpm2cpio logrotate-1.0-1.i386.rpm |cpio -ivd usr/man/man8/logrotate.8
usr/man/man8/logrotate.8
14 blocks
#
```

In this case, the `cpio` options `-i`, `-v`, and `-d` direct `cpio` to:

- Extract one or more files from an archive.
- Display the names of any files processed, along with the size of the archive file, in 512-byte blocks.[1]
- Create any directories that precede the filename specified in the `cpio` command.

So where did the file end up? The last option (`-d`) to `cpio` provides a hint. Let's take a look:

```
# ls -al
total 5
-rw-rw-r-- 1 root root 3918 May 30 11:02 logrotate-1.0-1.i386.rpm
drwx------ 3 root root 1024 Jul 14 12:42 usr
# cd usr
# ls -al
total 1
drwx------ 3 root root 1024 Jul 14 12:42 man
# cd man
# ls -al
total 1
drwx------ 2 root root 1024 Jul 14 12:42 man8
# cd man8
# ls -al
total 1
-rw-r--r-- 1 root root 706 Jul 14 12:42 logrotate.8
# cat logrotate.8
.\" logrotate - log file rotator
.TH rpm 8 "28 November 1995" "Red Hat Software" "Red Hat Linux"
.SH NAME
logrotate \- log file rotator
.SH SYNOPSIS
\fBlogrotate\fP [configfiles]
.SH DESCRIPTION
\fBlogrotate\fP is a tool to prevent log files from growing without
```

⋮

Since the current directory didn't have a `usr/man/man8/` path in it, the `-d` option caused `cpio` to create all the directories leading up to the `logrotate.8` file in the current directory. Based on this, it's probably safest to use `cpio` *outside* the normal system directories unless you're comfortable with `cpio`, and you know what you're doing!

[1] Note that the size displayed by `cpio` is the size of the `cpio` archive and not the package file.

8.3 Source Package Files and How To Use Them

One day, you may run across a package file with a name similar to the following:

`etcskel-1.0-3.`*`src`*`.rpm`

Notice the `src`. Is that a new kind of computer? If you use RPM on an Intel-based computer, you'd normally expect to find `i386` there. Maybe someone messed up the name of the file. Well, we know that the `file` command can display information about a package file, even if the filename has been changed. We've used it before to figure out what package a file contains:

```
# file foo.bar
foo.bar:  RPM v2 bin i386 eject-1.2-2
#
```

In this example, `foo.bar` is an RPM version 2 file, containing an executable package – hence, the "`bin`" – built for Intel processors – the "`i386`". The package is eject version 1.2, release 2.

Let's try the `file` command on this mystery file and see what we can find out about it:

```
# file etcskel-1.0-3.src.rpm
etcskel-1.0-3.src.rpm:  RPM v2 src i386 etcskel-1.0-3
#
```

Well, it's a package file all right – for version 1.0, release 3 of the etcskel package. It's in RPM version 2 format, and built for Intel-based systems. But what does the "`src`" mean?

8.3.1 A gentle introduction to source code

This package file contains not the executable, or "binary", files that a normal package contains, but rather the "source" files required to create those binaries. When programmers create a new program, they write the instructions, often called "code", in one or more files. The source code is then compiled into a binary that can be executed by the computer.

As part of the process of building package files (a process we cover in great detail in the second half of this book), two types of package files are created:

1. The binary, or executable, package file
2. The source package file

The source package contains everything needed to recreate not only the programs and associated files that are contained in the binary package file, but the binary and source package files themselves.

8.3.2 Do you *really* need more information than this?

The following discussion is going to get rather technical. Unless you're the type of person who likes to take other people's code and modify it, chances are you won't need much more information than this. But if you're still interested, let's explore further.

8.3.3 So what can I do with it?

In the case of source package files, one of the things that can be done with them is that they can be installed. Let's try an install of a source package:

```
# rpm -i cdp-0.33-3.src.rpm
#
```

Well that doesn't tell us very much and, take our word for it, adding `-v` doesn't improve the situation appreciably. Let's haul out the big guns and try `-vv`:

```
# rpm -ivv cdp-0.33-3.src.rpm
D: installing cdp-0.33-3.src.rpm
Installing cdp-0.33-3.src.rpm
D: package is a source package major = 2
D: installing a source package
D: sources in:  ///usr/src/redhat/SOURCES
D: spec file in:  ///usr/src/redhat/SPECS
D: file "cdp-0.33-cdplay.patch" complete
D: file "cdp-0.33-fsstnd.patch" complete
D: file "cdp-0.33.spec" complete
D: file "cdp-0.33.tgz" complete
D: renaming ///usr/src/redhat/SOURCES/cdp-0.33.spec
to
///usr/src/redhat/SPECS/cdp-0.33.spec
#
```

What does this output tell us? Well, RPM recognizes that the file is a source package. It mentions that sources (we know what *they* are) are in `/usr/src/redhat/SOURCES`. Let's take a look:

```
# ls -al /usr/src/redhat/SOURCES/
-rw-rw-r-- 1 root root 364 Apr 24 22:35 cdp-0.33-cdplay.patch
-rw-r--r-- 1 root root 916 Jan 8 12:07 cdp-0.33-fsstnd.patch
-rw-r--r-- 1 root root 148916 Nov 10 1995 cdp-0.33.tgz
#
```

There are some files that seem to be related to `cdp` there. The two files ending with

"`.patch`" are patches to the source. RPM permits patches to be processed when building binary packages. The patches are bundled along with the original, unmodified sources in the source package.

The last file is a gzipped tar file. If you've gotten software off the Internet, you're probably familiar with `tar` files, gzipped or not. If we look inside the file, we should see all the usual kinds of things: README files, a Makefile or two, and some source code:

```
# tar ztf cdp-0.33.tgz
cdp-0.33/COPYING
cdp-0.33/ChangeLog
cdp-0.33/INSTALL
cdp-0.33/Makefile
cdp-0.33/README
cdp-0.33/cdp
cdp-0.33/cdp-0.33.lsm
cdp-0.33/cdp.1
cdp-0.33/cdp.1.Z
cdp-0.33/cdp.c
cdp-0.33/cdp.h
```

There's more, but you get the idea. OK, so there are the sources. But what is that "spec" file mentioned in the output? It mentions something about "`/usr/src/redhat/SPECS`", so let's see what we have in that directory:

```
# ls -al /usr/src/redhat/SPECS
-rw-r--r-- 1 root root 397 Apr 24 22:36 cdp-0.33.spec
```

Without making a long story too short, a spec file contains information used by RPM to create the binary and source packages. Using the spec file, RPM:

- Unpacks the sources.
- Applies patches (if any exist).
- Builds the software.
- Creates the binary package file.
- Creates the source package file.
- Cleans up after itself.

The neatest part of this is that RPM does this all automatically, under the control of the spec file. That's about all we're going to say about how RPM builds packages. For more information, please refer to the second half of this book.

8.3.4 Stick with us!

As we've noted several times, we'll be covering the entire subject of building packages with RPM, in the second half of the book. Be forewarned, however: Package building, while straightforward, is not a task for people new to programming. But if you've written a program or two, you'll probably find RPM's package building a piece of cake.

Part II

RPM and Developers – How to Distribute Your Software More Easily With RPM

Chapter 9

The Philosophy Behind RPM

As we saw in the first half of this book, RPM can make life much easier for the user. With automated installs, upgrades, and erasures, RPM can take a lot of the guesswork out of keeping a computer system up-to-date.

But what about people that sling code for a living? Does RPM have anything to offer them? The answer is *yes*! One of the best things about RPM is that although it was designed to make life easier for users, it was written by people that would be using it to build *many* packages. So the design philosophy of RPM has a definite bias toward making life easier for developers. Here are some of the reasons you should consider building packages with RPM:

9.1 Pristine Sources

While many developers might use RPM to package their own software, just as many, if not more, are going to be packaging software that they have not written. Because of this, there are some aspects to RPM's design that are geared toward "third-party" package builders. One such aspect is RPM's use of "pristine" sources.

When a third-party package builder decides to package someone else's software, they often get the software from the Net, normally as a `tar` file compressed with something like GNU zip. That's probably about the only generalization we can make when talking about software that is eligible for packaging. Once we look inside the `tar` file, there are a world of possible differences:

- The application could be available in pure source form, in pure binary form, or some combination of both.
- The application might have been written to be built using `make`, `imake`, or a script included with the sources. Or, it might have to be built entirely by hand.
- The application might need to be configured prior to use. Maybe it uses GNU `configure`, a custom configuration script, or one or more files that need to be edited to reflect the target environment.

- The application might have been written to reside in specific directories, and those directories do not exist, or are not appropriate on the target system.
- The application might not even support the target environment, requiring all manner of changes to port it to the target environment.

We could go on, but you probably get the idea. It's a rare application that comes off the Net ready to package, and the changes required vary widely. What to do?

This is where the concept of pristine sources comes in. RPM has been designed to use the sources as they come from the application's developer, no matter how it has been packaged and configured. The main benefit is that the changes you as a package builder need to make, remain separate from the original sources, in a distinct collection of patches.

This may not sound like much of an advantage, but consider how this would work if a new version of the application came out. If the new version had a few localized bug fixes, it's entirely possible the original patches could be applied, and a new package built, with a single RPM command. Even if the patches didn't apply cleanly, it would at least give an indication as to what might need to be done to get the new version built and packaged.

If your users sometimes customize packages, having pristine sources makes it easier for them, too. They can see what patches you've created and can easily add their own.

Another benefit to using pristine sources is that it makes keeping track of multiple versions of a package simple. Instead of keeping patched sources around, or battling a revision control system, it's only necessary to keep:

- The original sources in their tar file.
- A copy of the patches you applied to get the application to build.
- A file used by RPM to control the package building process.

With these three items, it's possible to easily build the package at any time. Keeping track of multiple versions only entails keeping track of each version of these three components, rather than hundreds or thousands of patched source files.

In fact, it gets better than that. RPM can also build a source package containing these three components. The source package, named using RPM's standard naming convention, keeps everything you need to recreate a specific version of a package, in one uniquely named file. Keeping track of multiple versions of multiple packages is simply a matter of keeping the appropriate source packages around. Everything else can be built from them.

9.2 Easy Builds

RPM makes it easy to build packages. Just as with the use of pristine sources, the fact that the build process is simple is an even greater advantage to the third-party package builders responsible for many packages, than it is to a one-package software development house. But in either case, RPM's ease of building is a welcome relief. The following sections document some of the ways that RPM makes building packages a straightforward process.

9.2.1 Reproducible Builds

One of the biggest problems facing developers is reproducing a particular build. This single problem is the main reason so much effort is put into creating and deploying version control systems to manage sources.

While RPM cannot compete with a full-blown revision control system, it does an excellent job of keeping in one place everything required to build a particular version of a package. Remember the source package we mentioned above? With one command, RPM can open the package, extract the sources, patch them, perform a build, and create a new binary package, ready for your users. The best part is that the binary package will be the same *every* time you build it because everything needed to create it is kept in one source package.

9.2.2 Unattended Builds

As we mentioned above, completely building a package takes only one RPM command. This makes it easy to set up automated build procedures that can build one hundred packages as easily as one. Anything from a single package consisting of one application to the several hundred packages that comprise an entire operating system, can be built automatically using RPM.

9.3 Multi-architecture/operating system Support

It has always been a fact of life for software developers that their applications may need to be ported to multiple operating systems. It is also becoming more common that a particular operating system might run on several different platforms, or architectures.

RPM's ability to support multiple architectures and operating systems makes it easy to build the same package for many OS/platform combinations. A package may be configured to build on only one architecture/OS combination, or on several. The only limitation is the application's portability.

9.4 Easier For Your Users

While we are primarily concerned with RPM's advantages from the developer's point of view, it's worth looking at RPM from the user's standpoint for a moment. After all, if RPM makes life easier for your users, that can translate into lower support costs.

9.4.1 Easy Upgrades

Probably the biggest headache for user and developer alike is the upgrade of an application, or worse yet, an entire operating system! RPM can make upgrading a one-step process. With one command, a new package can be installed, and the remnants of the old package removed.

9.4.2 Intelligent Configuration File Handling

Configuration files — nearly every application has them. They may go by different names, but they all control the behavior of their application. Users normally customize config files to their liking and would be upset if their customizations were lost during the installation, upgrade, or removal of a package.

RPM takes special care with a user's config files. By using MD5 checksums, RPM can determine what action is most appropriate with a config file. If a config file has been modified by the user and has to be replaced, it is saved. That way a user's modifications are never lost.

9.4.3 Powerful Query Capabilities

RPM uses a database to keep track of all files it installs. RPM's database provides other benefits, such as the wide variety of information that can be easily retrieved from it. RPM's query command makes it easy for users to quickly answer a number of questions, such as:

- Where did this file come from? Is it part of a package?
- What does this package do?
- What packages are installed on my system?

These are just a few examples of the many ways RPM can provide information about one or more packages on a user's system.

9.4.4 Easy Package Verification

Another way that RPM leverages the information stored in its database, is by providing an easy way to verify that a package is properly installed. With this capability, RPM makes it easy to determine, for example, what packages were damaged by a wildcard delete in `/usr/bin`. In addition, RPM's verification command can detect changes to file attributes, such as a file's permissions, ownership, and size.

9.5 To Summarize...

RPM was written *by* developers *for* developers. It makes building packages as easy as possible, even if the software being packaged hasn't been developed in-house. In addition, RPM presents some significant advantages to users, thereby reducing support needs.

In the next chapter, we'll introduce the basic concepts of package building with RPM.

Chapter 10

The Basics of Developing With RPM

Now that we've seen the design philosophy of RPM, let's look at the nuts and bolts of RPM's build process. Building a package is similar to compiling code – there are inputs, an engine that does the dirty work, and outputs.

10.1 The Inputs

There are three different kinds of inputs that are used to drive RPM's build process. Two of the three inputs are required, and the third, strictly speaking, is optional. But unless you're packaging your own code, chances are you'll need it.

10.1.1 The Sources

First and foremost, are the sources. After all, without them, there wouldn't be much to build! In the case of packaging someone else's software, the sources should be kept as the author distributed them, which usually means a compressed `tar` file. RPM can handle other archive formats, but a bit more up-front effort is required.

In any case, you should not modify the sources used in the package building process. If you're a third-party package builder, that means the sources should be just the way you got them from the author's FTP site. If it's your own software, the choice is up to you, but you should consider starting with your mainstream sources.

10.1.2 The Patches

Why all the emphasis on unmodified sources? Because RPM gives you the ability to automatically apply patches to them. Usually, the nature of these patches falls into one of the following categories:

- The patch addresses an issue specific to the target system. This could include changing makefiles to install the application into the appropriate directories, or resolving cross-platform conflicts, such as replacing BSD system calls with their SYSV counterparts.
- The patch creates files that are normally created during a configuration step in the installation process. Many times, it's necessary to either edit configuration files or scripts in order to set things up for compilation. In other cases, a configuration utility needs to be run before the sources are compiled. In either instance, the patches create the environment required for proper compilation.

Creating the Patches

While it might sound a bit daunting to take into account the types of patches outlined above, it's really quite simple. Here's how it's done:

1. Unpack the sources.
2. Rename the top-level directory. Make it end with ".orig", for example.
3. Unpack the sources again, leaving the top-level directory name unchanged.

The source tree that you created the second time will be the one you'll use to get the software to build.

If the software builds with no changes required, that's great – you won't need a patch. But if you had to make any changes, you'll have to create a set of patches. To do so, simply clean the source directory of any binaries. Then, issue a recursive `diff` command to compare the source tree you used for the build, against the original, unmodified source tree. It's as easy as that!

10.1.3 The Spec File

The spec file is at the heart of RPM's packaging building process. Similar in concept to a makefile, it contains information required by RPM to build the package, as well as instructions telling RPM *how* to build it. The spec file also dictates exactly what files are a part of the package, and where they should be installed.

As you might imagine, with this many responsibilities, the spec file format can be a bit complex. However, it's broken into several sections, making it easier to handle. All told, there are eight sections:

The Preamble The preamble contains information that will be displayed when users request information about the package. This would include a description of the package's function, the version number of the software, and so on. Also contained in the preamble are lines identifying sources, patches, and even an icon to be used if the package is manipulated by graphical interface.

The Prep Section The prep section is where the actual work of building a package starts. As the name implies, this section is where the necessary preparations are made prior to the actual building of the software. In general, if anything needs to be done to the sources prior to building the software, it needs to happen in the prep section. Usually, this boils down to unpacking the sources.

The contents of this section are an ordinary shell script. However, RPM does provide two macros to make life easier. One macro can unpack a compressed `tar` file and `cd` into the source directory. The other macro easily applies patches to the unpacked sources.

The Build Section Like the prep section, the build section consists of a shell script. As you might guess, this section is used to perform whatever commands are required to actually compile the sources. This section could consist of a single `make` command, or be more complex if the build process requires it. Since most software is built today using `make`, there are no macros available in this section.

The Install Section Also containing a shell script, the install section is used to perform the commands required to actually install the software. If the software's author added an install target in the makefile, this section might only consist of a `make install` command. Otherwise, you'll need to add the usual assortment of `cp`, `mv`, or `install` commands to get the job done.

Install and Uninstall Scripts While the previous sections contained either information required by RPM to build the package, or the actual commands to do the deed, this section is different. It consists of scripts that will be run, *on the user's system*, when the package is actually installed or removed. RPM can execute a script:

- Prior to the package being installed.
- After the package has been installed.
- Prior to the package being erased.
- After the package has been erased.

One example of when this capability would be required is when a package contains shared libraries. In this case, `ldconfig` would need to be run after the package is installed or erased. As another example, if a package contains a shell, the file `/etc/shells` would need to be updated appropriately when the package was installed or erased.

The Verify Script This is another script that is executed on the user's system. It is executed when RPM verifies the package's proper installation. While RPM does most of the work verifying packages, this script can be used to verify aspects of the package that are beyond RPM's capabilities.

The Clean Section Another script that can be present is a script that can clean things up after the build. This script is rarely used, since RPM normally does a good job of clean-up in most build environments.

The File List The last section consists of a list of files that will comprise the package. Additionally, a number of macros can be used to control file attributes when installed, as well as to denote which files are documentation, and which contain configuration information. The file list is very important — if it is missing, no package will be built.

10.2 The Engine: RPM

At the center of the action is RPM. It performs a number of steps during the build process:

- Executes the commands and macros in the prep section of the spec file.
- Checks the contents of the file list.
- Executes the commands and macros in the build section of the spec file.
- Executes the commands and macros in the install section of the spec file. Any macros in the file list are executed at this time, too.
- Creates the binary package file.
- Creates the source package file.

By using different options on the RPM command line, the build process can be stopped at any of the steps above. This makes the initial building of a package that much easier, as it is then possible to see whether each step completed successfully before continuing on to the next step.

10.3 The Outputs

The end product of this entire process is a source package file and a binary package file.

10.3.1 The Source Package File

The source package file is a specially formatted archive that contains the following files:

- The original compressed `tar` file(s).
- The spec file.
- The patches.

Since the source package contains everything needed to create the binary package, the source package, *and* provide the original sources, it's a great way to distribute source code. As mentioned earlier, it's also a great way to archive all the information needed to rebuild a particular version of the package.

10.3.2 The Binary RPM

The binary package file is the one part of the entire RPM building process that is most visible to the user. It contains the files that comprise the application, along with any additional information needed to install and erase it. The binary package file is where the "rubber hits the road."

10.4 And Now...

Now that we've seen, in broad brush terms, the way RPM builds packages, let's take a look at an actual build. The next chapter will do just that, showing how simple it can be to build a package.

Chapter 11

Building Packages: A Simple Example

In the previous chapter, we looked at RPM's build process from a conceptual level. In this chapter, we will be performing an actual build using RPM. In order to keep things understandable for this first pass, the build will be very simple. Once we've covered the basics, we'll present more real-world examples in later chapters.

11.1 Creating the Build Directory Structure

RPM requires a set of directories in which to perform the build. While the directories' locations and names can be changed, unless there's a reason to do so, it's best to use the default layout. Note that if you've installed RPM, the build directories are most likely in place already.

The normal directory layout consists of a single top-level directory (The default name is `/usr/src/redhat`), with five subdirectories. The five subdirectories and their functions are:

- `/usr/src/redhat/SOURCES` — Contains the original sources, patches, and icon files.
- `/usr/src/redhat/SPECS` — Contains the spec files used to control the build process.
- `/usr/src/redhat/BUILD` — The directory in which the sources are unpacked, and the software is built.
- `/usr/src/redhat/RPMS` — Contains the binary package files created by the build process.
- `/usr/src/redhat/SRPMS` — Contains the source package files created by the build process.

In general, there are no special requirements that need to be met when creating these directories. In fact, the only important requirement is that the `BUILD` directory be part of a filesystem with sufficient free space to build the largest package expected. Here is a directory listing showing a typical build directory tree:

```
# ls -lF /usr/src/redhat
total 5
drwxr-xr-x 3 root root 1024 Aug 5 13:12 BUILD/
drwxr-xr-x 3 root root 1024 Jul 17 17:51 RPMS/
drwxr-xr-x 4 root root 1024 Aug 4 22:31 SOURCES/
drwxr-xr-x 2 root root 1024 Aug 5 13:12 SPECS/
drwxr-xr-x 2 root root 1024 Aug 4 22:28 SRPMS/
#
```

Now that we have the directories ready to go, it's time to prepare for the build. For the remainder of this chapter, we'll be building a fictional piece of software known as `cdplayer`.[1]

11.2 Getting the Sources

The first thing we need to do in order to build a package for `cdplayer`, is to obtain the sources. Being avid `cdplayer` fans from way back, we know that the latest source can be found at GnomoVision's FTP site, so we go get a copy.

We now have a gzipped `tar` file of `cdplayer` version 1.0 on our system. After putting a copy in the `SOURCES` directory, we're ready to tell RPM what to do with it.

11.3 Creating the Spec File

The way we direct RPM in the build process is to create a spec file. As we saw in the previous chapter, the spec file contains eight different sections, most of which are required. Let's go through each section and create `cdplayer`'s spec file as we go.

11.3.1 The Preamble

The preamble contains a wealth of information about the package being built, and the people that built it. Here's `cdplayer`'s preamble:

```
#
# Example spec file for cdplayer app...
```

[1] In reality, this software is a mercilessly hacked version of `cdp`, which was written by Sariel Har-Peled. The software was hacked to provide a simple example package, and in no way represents the fine work done by Sariel on `cdp`.

```
#
Summary:  A CD player app that rocks!
Name:  cdplayer
Version:  1.0
Release:  1
Copyright:  GPL
Group:  Applications/Sound
Source:  ftp://ftp.gnomovision.com/pub/cdplayer/cdplayer-1.0.tgz
URL: http://www.gnomovision.com/cdplayer/cdplayer.html
Distribution:  WSS Linux
Vendor:  White Socks Software, Inc.
Packager:  Santa Claus <sclaus@northpole.com>

%description
It slices!  It dices!  It's a CD player app that
can't be beat.  By using the resonant frequency
of the CD itself, it is able to simulate 20X
oversampling.  This leads to sound quality that
cannot be equaled with more mundane software...
```

In general, the preamble consists of entries, one per line, that start with a *tag* followed by a colon, and then some information. For example, the line starting with "`Summary:`" gives a short description of the packaged software that can be displayed by RPM. The order of the lines is not important, as long as they appear in the preamble.

Let's take a look at each line and see what function it performs:

Name — The `name` line defines what the package will actually be called. In general, it's a good idea to use the name of the software. The name will also be included in the package label, and the package filename.

Version — The `version` line should be set to the version of the software being packaged. The version will also be included in the package label, and the package filename.

Release — The `release` is a number that is used to represent the number of times the software, at the present version, has been packaged. You can think of it as the package's version number. The release is also part of the package label and package filename.

Copyright — The `copyright` line is used to hold the packaged software's copyright information. This makes it easy to determine which packages can be freely redistributed, and which cannot. In our case, `cdplayer` is made available under the terms of the GNU General Public License, so we've put `GPL` on the line.

Group — The `group` line is used to hold a string that defines how the packaged software should be grouped with other packages. The string consists of a series of words separated by slashes. From left to right, the words describe the packaged software more explicitly. We grouped `cdplayer` under `Applications`, because it is an application, and then under `Sound`, since it is an application that is sound-related.

Source — The `source` line serves two purposes:

- To document where the packaged software's sources can be found.
- To give the name of the source file as it exists in the SOURCES subdirectory.

In our example, the `cdplayer` sources are contained in the file `cdplayer-1.0.tgz`, which is available from `ftp.gnomovision.com`, in the directory `/pub/cdplayer`. RPM actually ignores everything prior to the last filename in the source line, so the first part of the source string could be anything you'd like. Traditionally, the `source` line usually contains a Uniform Resource Locator, or URL.

URL — The `URL` line is used to contain a URL, like the `source` line. How are they different? While the `source` line is used to provide the source filename to RPM, the `URL` line points to *documentation* for the software being packaged.

Distribution — The `distribution` line contains the name of the product which the packaged software is a part of. In the Linux world, the operating system is often packaged together into a "distribution", hence the name. Since we're using a fictional application in this example, we've filled in the distribution line with the name of a fictional distribution. There's no requirement that the spec file contain a `distribution` line, so individuals will probably omit this.

Vendor — The `vendor` line identifies the organization that distributes the software. Maintaining our fictional motif, we've invented fictional company, White Socks Software, to add to our spec file. Individuals will probably omit this as well.

Packager — The `packager` line is used to identify the organization that actually *packaged* the software, as opposed to the author of the software. In our example, we've chosen the greatest packager of them all, Santa Claus, to work at White Socks Software. Note that we've included contact information, in the form of an e-mail address.

Description — The `description` line is a bit different, in that it starts with a percent sign. It is also different because the information can take up more than one line. It is used to provide a more detailed description of the packaged software than the `summary` line.

A Comment on Comments — At the top of the spec file, there are three lines, each starting with a pound sign. These are comments and can be sprinkled throughout the spec file to make it more easily understood.

11.3.2 The %prep Section

With the preamble, we provided a wealth of information. The majority of this information is meant for human consumption. Only the `name`, `version`, `release`, and `source` lines have a direct bearing on the package building process. However, in the `%prep` section, the focus is entirely on directing RPM through the process of preparing the software for building.

It is in the `%prep` section that the build environment for the software is created, starting with removing the remnants of any previous builds. Following this, the source archive is expanded. Here is what the `%prep` section looks like in our example spec file:

```
%prep
rm -rf $RPM_BUILD_DIR/cdplayer-1.0
zcat $RPM_SOURCE_DIR/cdplayer-1.0.tgz | tar -xvf -
```

If the `%prep` section looks like a script, that's because it is. Any `sh` constructs can be used here, including expansion of environment variables (Like the `$RPM_BUILD_DIR` variable defined by RPM), and piping the output of `zcat` through `tar`.[2]

In this case, we perform a recursive delete in the build directory to remove any old builds. We then uncompress the gzipped `tar` file, and extract its contents into the build directory.

Quite often, the sources may require patching in order to build properly. The `%prep` section is the appropriate place to patch the sources, but in this example, no patching is required. Fear not, however, as we'll explore patching in all its glory in chapter 20 on page 279, when we build a more complex package.

Making Life Easier With Macros

While the `%prep` section as we've described it isn't *that* difficult to understand, RPM provides macros to make life even easier. In this simple example, there's precious little that can be made easier, but macros will prevent a wealth of headaches when it's time to build more complex packages. The macro we'll introduce here is the `%setup` macro.

The average gzipped `tar` file is `%setup`'s stock in trade. Like the hand-crafted `%prep` section we described above, it cleans up old build trees and then uncompresses and extracts the files from the original source. While `%setup` has a number of options that we'll cover in later chapters, for now all we need for a `%prep` section is:

```
%prep
%setup
```

[2]For more information on the environment variables used in the build-time scripts, please refer to section 13.3.1 on page 180.

That is simpler than our `%prep` section, so let's use the `%setup` macro instead. The `%setup` macro has a number of options to handle many different situations. For more information on this and other macros, please see section 13.4 on page 186.

In our example here, the `%prep` section is complete. Next comes the actual build.

11.3.3 The `%build` Section

Not surprisingly, the part of the spec file that is responsible for performing the build, is the `%build` section. Like the `%prep` section, the `%build` section is an ordinary `sh` script. Unlike the `%prep` section, there are no macros. The reason for this is that the process of building software is either going to be very easy, or highly complicated. In either case, macros won't help much. In our example, the build process is simple:

```
%build
make
```

Thanks to the `make` utility, only one command is necessary to build the `cdplayer` application. In the case of an application with more esoteric build requirements, the `%build` section could get a bit more interesting.

11.3.4 The `%install` Section

The `%install` section is executed as a `sh` script, just like `%prep` and `%build`. If the application is built with `make` and has an "install" target, the `%install` section will also be straightforward. The `cdplayer` application is a good example of this:

```
%install
make install
```

If the application doesn't have a means of automatically installing itself, it will be necessary to create a script to do so, and place it in the `%install` section.

11.3.5 The `%files` Section

The `%files` section is different from the others, in that it contains a list of the files that are part of the package. Always remember — if it isn't in the file list, it won't be put in the package!

```
%files
%doc README
/usr/local/bin/cdp
/usr/local/bin/cdplay
/usr/local/man/man1/cdp.1
```

The line starting with `%doc` is an example of RPM's handling of different file types. As you might guess, `%doc` stands for documentation. The `%doc` directive is used to mark files as being documentation. In the example above, the `README` file will be placed in a package-specific directory, located in `/usr/doc`, and called `cdplayer-1.0-1`. It's also possible to mark files as documentation and have them installed in other directories. This is covered in more detail in section 13.6.1 on page 199.

The rest of the files in the example are shown with complete paths. This is necessary as the files will actually be installed in those directories by the application's makefile. Since RPM needs to be able to find the files prior to packaging them, complete paths are required.

How Do You Create the File List?

Since RPM automates so many aspects of software installation, it's easy to fall into the trap of assuming that RPM does *everything* for you. Not so! One task that is still a manual process is creating the file list. While it may seem at first glance, that it could be automated somehow, it's actually a more difficult problem than it seems.

Since the majority of an application's files are installed by its makefile, RPM has no control over that part of the build process, and therefore, cannot automatically determine which files should be part of the package. Some people have attempted to use a modified version of `install` that logs the name of every file it installs. But not every makefile uses `install`, or if it does, uses it sporadically.

Another approach tried was to obtain a list of every file on the build system, immediately before and after a build, and use the differences as the file list. While this approach will certainly find every file that the application installed, it can also pick up extraneous files, such as system logs, files in `/tmp`, and the like. The only way to begin to make this approach workable would be to do nothing else on the build system, which is highly inconvenient. This approach also precludes building more than one package on the system at any given time.

At present, the best way to create the file list is to read the makefile to see what files it installs, verify this against the files installed on the build system, and create the list.

11.3.6 The Missing Spec File Sections

Since our example spec file is somewhat simplistic, it's missing two sections that might be used in more complex situations. We'll go over each one briefly here. More complete information on these sections will be covered at various points in the book.

The Install/Uninstall Scripts

One missing section to our spec file is the section that would define one or more of four possible scripts. The scripts are executed at various times when a package is installed or erased.

The scripts can be executed:

- Before a package is installed.

- After a package is installed.
- Before a package is erased.
- After a package is erased.

We'll see how these scripts are used in chapter 20 on page 279.

The `%clean` Section

The other missing section has the rather descriptive title of `%clean`. This section can be used to clean up any files that are not part of the application's normal build area. For example, if the application creates a directory structure in `/tmp` as part of its build, it will not be removed. By adding a `sh` script to the `%clean` section, such situations can be handled gracefully, right after the binary package is created.

11.4 Starting the Build

Now it's time to begin the build. First, we change directory into the directory holding `cdplayer`'s spec file:

```
# cd /usr/src/redhat/SPECS
#
```

Next, we start the build with an `rpm -b` command:

```
# rpm -ba cdplayer-1.0.spec
```

The `a` following the `-b` option directs RPM to perform all phases of the build process. Sometimes it is necessary to stop at various phases during the initial build to resolve problems that crop up while writing the spec file. In these cases, other letters can be used after the `-b` in order to stop the build at the desired phase. For this example however, we will continue through the entire build process.

In this example, the only other argument to the build command is the name of the package's spec file. This can be wild-carded to build more than one package, but in our example, we'll stick with one.

Let's look at RPM's output during the build:

```
* Package: cdplayer
+ umask 022
+ echo Excuting: %prep
Excuting: %prep
+ cd /usr/src/redhat/BUILD
```

```
+ cd /usr/src/redhat/BUILD
+ rm -rf cdplayer-1.0
+ gzip -dc /usr/src/redhat/SOURCES/cdplayer-1.0.tgz
+ tar -xvvf -
drwxrwxr-x root/users        0 Aug  4 22:30 1996 cdplayer-1.0/
-rw-r--r-- root/users    17982 Nov 10 01:10 1995 cdplayer-1.0/COPYING
-rw-r--r-- root/users      627 Nov 10 01:10 1995 cdplayer-1.0/ChangeLog
-rw-r--r-- root/users      482 Nov 10 01:11 1995 cdplayer-1.0/INSTALL
.
.
.
-rw-r--r-- root/users     2720 Nov 10 01:10 1995 cdplayer-1.0/struct.h
-rw-r--r-- root/users      730 Nov 10 01:10 1995 cdplayer-1.0/vol.c
-rw-r--r-- root/users     2806 Nov 10 01:10 1995 cdplayer-1.0/volume.c
-rw-r--r-- root/users     1515 Nov 10 01:10 1995 cdplayer-1.0/volume.h
+ [ 0 -ne 0 ]
+ cd cdplayer-1.0
+ cd /usr/src/redhat/BUILD/cdplayer-1.0
+ chown -R root.root .
+ chmod -R a+rX,g-w,o-w .
+ exit 0
```

The output continues, but let's stop here for a moment, and discuss what has happened so far.

At the start of the output, RPM displays the package name (`cdplayer`), sets the umask, and starts executing the `%prep` section. Thanks to the `%setup` macro, RPM then changes directory into the build area, removes any existing old sources, and extracts the sources from the original compressed tar file. Although each file is listed as it is extracted, we've omitted most of the files listed, to save space.

The `%setup` macro continues by changing directory into `cdplayer`'s top-level source directory and setting the file ownership and permissions properly. As you can see, it does quite a bit of work for you.

Let's take a look at the output from the `%build` section next:

```
+ umask 022
+ echo Excuting: %build
Excuting: %build
+ cd /usr/src/redhat/BUILD
+ cd cdplayer-1.0
+ make
gcc -Wall -O2  -c -I/usr/include/ncurses  cdp.c
gcc -Wall -O2  -c -I/usr/include/ncurses  color.c
gcc -Wall -O2  -c -I/usr/include/ncurses  display.c
gcc -Wall -O2  -c -I/usr/include/ncurses  misc.c
gcc -Wall -O2  -c -I/usr/include/ncurses  volume.c
volume.c: In function 'mix_set_volume':
```

```
volume.c:67: warning: implicit declaration of function 'ioctl'
gcc -Wall -O2  -c -I/usr/include/ncurses  hardware.c
gcc -Wall -O2  -c -I/usr/include/ncurses  database.c
gcc -Wall -O2  -c -I/usr/include/ncurses  getline.c
gcc -o cdp cdp.o color.o display.o misc.o volume.o hardware.o database.o
getline.o  -I/usr/include/ncurses  -L/usr/lib -lncurses
groff -Tascii -man cdp.1 | compress >cdp.1.Z
+ exit 0
```

There are no surprises here. After setting the umask and changing directory into `cdplayer`'s top-level directory, RPM issues the `make` command we put into the spec file. The rest of the output comes from `make` as it actually builds the software. Next comes the `%install` section:

```
+ umask 022
+ echo Excuting: %install
Excuting: %install
+ cd /usr/src/redhat/BUILD
+ cd cdplayer-1.0
+ make install
chmod 755 cdp
chmod 644 cdp.1.Z
cp cdp /usr/local/bin
ln -s /usr/local/bin/cdp /usr/local/bin/cdplay
cp cdp.1 /usr/local/man/man1
+ exit 0
```

Just like the previous sections, RPM again sets the umask and changes directory into the proper directory. It then executes `cdplayer`'s install target, installing the newly built software on the build system. Those of you that carefully studied the spec file might have noticed that the `README` file is not part of the install section. It's not a problem, as we see here:

```
+ umask 022
+ echo Excuting: special doc
Excuting: special doc
+ cd /usr/src/redhat/BUILD
+ cd cdplayer-1.0
+ DOCDIR=//usr/doc/cdplayer-1.0-1
+ rm -rf //usr/doc/cdplayer-1.0-1
+ mkdir -p //usr/doc/cdplayer-1.0-1
+ cp -ar README //usr/doc/cdplayer-1.0-1
+ exit 0
```

After the customary `umask` and `cd` commands, RPM constructs the path that will be used for `cdplayer`'s documentation directory. It then cleans out any preexisting directory and copies the README file into it. The cdplayer app is now installed on the build system. The only thing left to do is to create the actual package files, and perform some housekeeping. The binary package file is created first:

```
Binary Packaging: cdplayer-1.0-1
Finding dependencies...
Requires (2): libc.so.5 libncurses.so.2.0
usr/doc/cdplayer-1.0-1
usr/doc/cdplayer-1.0-1/README
usr/local/bin/cdp
usr/local/bin/cdplay
usr/local/man/man1/cdp.1
93 blocks
Generating signature: 0
Wrote: /usr/src/redhat/RPMS/i386/cdplayer-1.0-1.i386.rpm
```

The first line says it all: RPM is creating the binary package for `cdplayer` version 1.0, release 1. Next, RPM determines what packages are required by `cdplayer-1.0-1`. Part of this process entails running `ldd` on each executable program in the package. In this example, the package requires the libraries `libc.so.5`, and `libncurses.so.2.0`. Other dependency information can be included in the spec file, but for our example we'll keep it simple.

Following the dependency information, there is a list of every directory and file included in the package. The list displayed is actually the output of `cpio`, which is the archiving software used by RPM to bundle the package's files. The "`93 blocks`" is also printed by `cpio`.

The line "`Generating signature: 0`" means that RPM has not been directed to add a PGP signature to the package file. During this time, however, RPM still adds two signatures that can be used to verify the size and the MD5 checksum of the package file. Finally, we see confirmation that RPM has created the binary package file.

At this point, the application has been built, and the application's files have been packaged. There is no longer any need for any files created during the build, so they may be removed. In the case of the sources extracted into RPM's build directory, we can see that, at worst, they will be removed the next time the package is built. But what if there *were* files that we needed to remove? Well, they could be deleted here, in the `%clean` section:

```
+ umask 022
+ echo Excuting: %clean
Excuting: %clean
+ cd /usr/src/redhat/BUILD
+ cd cdplayer-1.0
+ exit 0
```

In our example, there are no other files outside of the build directory that are created during `cdplayer`'s build, so we don't need to expend any additional effort to clean things up.

The very last step performed by RPM is to create the source package file:

```
Source Packaging: cdplayer-1.0-1
cdplayer-1.0.spec
cdplayer-1.0.tgz
80 blocks
Generating signature: 0
Wrote: /usr/src/redhat/SRPMS/cdplayer-1.0-1.src.rpm
#
```

This file includes everything needed to recreate a binary package file, as well as a copy of itself. In this example, the only files needed to do that are the original sources and the spec file. In cases where the original sources needed to be modified, the source package includes one or more patch files. As when the binary package was created, we see `cpio`'s output listing each file archived, along with the archive's block size.

Just like a binary package, a source package file can have a PGP signature attached to it. In our case, we see that a PGP signature was not attached. The last message from RPM is to confirm the creation of the source package. Let's take a look at the end products. First, the binary package:

```
# ls -lF /usr/src/redhat/RPMS/i386/cdplayer-1.0-1.i386.rpm
-rw-r--r-- 1 root root 24698 Aug 6 22:22 RPMS/i386/cdplayer-1.0-1.i386.rpm
#
```

Note that we built `cdplayer` on an Intel-based system, so RPM placed the binary package files in the `i386` subdirectory.

Next, the source package file:

```
# ls -lF /usr/src/redhat/SRPMS/cdplayer-1.0-1.src.rpm
-rw-r--r-- 1 root root 41380 Aug 6 22:22 SRPMS/cdplayer-1.0-1.src.rpm
#
```

Everything went perfectly — we now have binary and source package files ready to use. But sometimes things don't go so well.

11.5 When Things Go Wrong

This example is a bit of a fairy tale, in that it went perfectly the first time. In real life, it often takes several tries to get it right.

11.5.1 Problems During the Build

As we alluded to earlier in the chapter, RPM can stop at various points in the build process. This allows package builders to look through the build directory and make sure everything is proceeding properly. If there are problems, stopping during the build process permits them to see exactly what is going wrong, and where. Here is a list of points RPM can be stoped at during the build:

- After the `%prep` section.
- After doing some cursory checks on the `%files` list.
- After the `%build` section.
- After the `%install` section.
- After the binary package has been created.

In addition, there is also a method that permits the package builder to "short circuit" the build process and direct RPM to skip over the initial steps. This is handy when the application is not yet ready for packaging and needs some fine tuning. This way, once the package builds, installs, and operates properly, the required patches to the original sources can be created, and plugged into the package's spec file.

11.5.2 Testing Newly Built Packages

Of course, the fact that an application has been packaged successfully doesn't necessarily mean that it will operate correctly when the package is actually installed. Testing is required. In the case of our example, it's perfect and doesn't need such testing.[3] But here is how testing would proceed

The first step is to find a test system. If you thought of simply using the build system, *bzzzzt*, try again! Think about it — in the course of building the package, the build system actually had the application installed on it. That is how RPM gets the files that are to be packaged: by building the software, installing it, and grabbing copies of the installed files, which are found using the `%files` list.

Some of you dissenters that have read the first half of the book might be thinking, "Why not just install the package on the build system using the `--replacefiles` option? That way, it'll just blow away the files installed by the build process and replace them with the packaged files." Well, you folks get a *bzzzzt*, too! Here's why.

Say, for example, that the software you're packaging installs a bunch of files — maybe a hundred. What does this mean? Well for one thing, it means that the package's `%files` list is going to be quite large. For another thing, the sheer number of files makes it likely that you'll miss one or two. What would happen then?

When RPM builds the software, there's no problem: the software builds, and the application's makefile merrily installs all the files. The next step in RPM's build process is to collect the files by reading the `%files` list, and to add each file listed to a `cpio` archive. What happens to the files you've missed? Nothing — they aren't added to the package file, but they *are* on your build system, installed just where they should be.

[3] Like we said, it's a fairy tale!

Next, when the package is installed using `--replacefiles`, RPM dutifully installs each of the packaged files, replacing the ones originally installed on the build system. The missed files? They aren't overwritten by RPM since they weren't in the package. But they're still on disk, right where the application expects them to be! If you go to test the application then, it will find every file it needs. But not every file came from the package. Bad news! Using a different system on which the application had never been built is one sure way to test for missing files.

That wraps up our fictional build. Now that we have some experience with RPM's build process, we can take a more in-depth look at RPM's build command.

Chapter 12

rpm -b Command Reference

rpm -b<stage> *options* file1.spec ... fileN.spec		
<stage>		Page
p	Execute %prep	140
c	Execute %prep, %build	142
i	Execute %prep, %build, %install	143
b	Execute %prep, %build, %install. Package (bin)	144
a	Execute %prep, %build, %install. Package (bin, src)	145
l	Check %files list	146
Parameters		
spec1 ... specN	One or more .spec files	
Build—specific Options		Page
--short-circuit	Force build to start at particular stage (-bc, -bi only)	149
--test	Create, save build scripts for review	154
--clean	Clean up after build	155
--sign	Add a digital signature to the package	153
--buildroot *<root>*	Execute %install using *<root>* as the root	156
--buildarch *<arch>*	Perform build for the *<arch>* architecture	151
--buildos *<os>*	Perform build for the *<os>* operating system	152
--timecheck *<secs>*	Print a warning if files are over *<secs>* old	159
General Options		Page
-vv	Display debugging information	160
--quiet	Produce as little output as possible	161
--rcfile *<rcfile>*	Set alternate rpmrc file to *<rcfile>*	161

12.1 rpm -b — What Does it Do?

When RPM is invoked with the -b option, the process of building a package is started. The rest of the command will determine exactly what is to be built and how far the build should proceed. In this chapter, we'll explore every aspect of `rpm -b`.

An RPM build command must have two additional pieces of information, over and above "`rpm -b`":

1. The names of one or more spec files representing software to be packaged.
2. The desired stage at which the build is to stop.

As we discussed in chapter 10, the spec file is one of the inputs to RPM's build process. It contains the information necessary for RPM to perform the build and package the software.

There are a number of stages that RPM goes through during a build. By specifying that the build process is to stop at a certain stage, the package builder can monitor the build's progress, make any changes necessary, and restart the build. Let's start by looking at the various stages that can be specified in a build command.

12.1.1 `rpm -bp` — Execute `%prep`

The command `rpm -bp` directs RPM to execute the very first step in the build process. In the spec file, this step is labeled `%prep`. Every command in the `%prep` section will be executed when the `-bp` option is used.

Here's a simple `%prep` section from the spec file we used in chapter 11:

```
%prep
%setup
```

This `%prep` section consists of a single `%setup` macro. When using `rpm -bp` against this spec file, we can see exactly what `%setup` does:

```
# rpm -bp cdplayer-1.0.spec
* Package: cdplayer
+ umask 022
+ echo Executing: %prep
Executing: %prep
+ cd /usr/src/redhat/BUILD
+ cd /usr/src/redhat/BUILD
+ rm -rf cdplayer-1.0
+ gzip -dc /usr/src/redhat/SOURCES/cdplayer-1.0.tgz
+ tar -xvvf -
drwxrwxr-x root/users        0 Aug  4 22:30 1996 cdplayer-1.0/
-rw-r--r-- root/users    17982 Nov 10 01:10 1995 cdplayer-1.0/COPYING
-rw-r--r-- root/users      627 Nov 10 01:10 1995 cdplayer-1.0/ChangeLog
...
-rw-r--r-- root/users     2806 Nov 10 01:10 1995 cdplayer-1.0/volume.c
-rw-r--r-- root/users     1515 Nov 10 01:10 1995 cdplayer-1.0/volume.h
+ [ 0 -ne 0 ]
```

```
+ cd cdplayer-1.0
+ cd /usr/src/redhat/BUILD/cdplayer-1.0
+ chown -R root.root .
+ chmod -R a+rX,g-w,o-w .
+ exit 0
#
```

First, RPM confirms that the `cdplayer` package is the subject of this build. Then it sets the umask and starts executing the `%prep` section. At this point, the `%setup` macro is doing its thing. It changes directory into the build area and removes any old copies of `cdplayer`'s build tree.

Next, `%setup` unzips the sources and uses `tar` to create the build tree. We've removed the complete listing of files, but be prepared to see *lots* of output if the software being packaged is large.

Finally, `%setup` changes directory into `cdplayer`'s build tree and changes ownership and file permissions appropriately. The `exit 0` signifies the end of the `%prep` section, and therefore, the end of the `%setup` macro. Since we used the `-bp` option, RPM stopped at this point. Let's see what RPM left in the build area:

```
# cd /usr/src/redhat/BUILD
# ls -l
total 1
drwxr-xr-x 2 root root 1024 Aug 4 22:30 cdplayer-1.0
#
```

There's the top-level directory. Changing directory into `cdplayer-1.0`, we find the sources are ready to be built:

```
# cd cdplayer-1.0
# ls -lF

total 216
-rw-r--r--   1 root     root        17982 Nov 10  1995 COPYING
-rw-r--r--   1 root     root          627 Nov 10  1995 ChangeLog
...
-rw-r--r--   1 root     root         2806 Nov 10  1995 volume.c
-rw-r--r--   1 root     root         1515 Nov 10  1995 volume.h
#
```

We can see that `%setup`'s `chown` and `chmod` commands did what they were supposed to — the files are owned by root, with permissions set appropriately.

If not stopped by the `-bp` option, the next step in RPM's build process would be to build the software. RPM can also be stopped at the end of the `%build` section in the spec file. This is done by using the `-bc` option:

12.1.2 rpm -bc — Execute %prep, %build

When the -bc option is used during a build, RPM stops once the software has been built. In terms of the spec file, every command in the %build section will be executed. In the following example, we've removed the output from the %prep section to cut down on the redundant output, but keep in mind that it is executed nonetheless:

```
# rpm -bc cdplayer-1.0.spec
* Package: cdplayer
Executing: %prep
...
+ exit 0
Executing: %build
+ cd /usr/src/redhat/BUILD
+ cd cdplayer-1.0
+ make
gcc -Wall -O2  -c -I/usr/include/ncurses  cdp.c
gcc -Wall -O2  -c -I/usr/include/ncurses  color.c
gcc -Wall -O2  -c -I/usr/include/ncurses  display.c
gcc -Wall -O2  -c -I/usr/include/ncurses  misc.c
gcc -Wall -O2  -c -I/usr/include/ncurses  volume.c
volume.c: In function ‘mix_set_volume’:
volume.c:67: warning: implicit declaration of function ‘ioctl’
gcc -Wall -O2  -c -I/usr/include/ncurses  hardware.c
gcc -Wall -O2  -c -I/usr/include/ncurses  database.c
gcc -Wall -O2  -c -I/usr/include/ncurses  getline.c
gcc -o cdp cdp.o color.o display.o misc.o volume.o hardware.o database.o
getline.o  -I/usr/include/ncurses  -L/usr/lib -lncurses
groff -Tascii -man cdp.1 | compress >cdp.1.Z
+ exit 0
#
```

After the command, we see RPM executing the %prep section (which we've removed almost entirely). Next, RPM starts executing the contents of the %build section. In our example spec file, the %build section looks like this:

```
%build
make
```

We see that prior to the make command, RPM changes directory into cdplayer's top-level directory. RPM then starts the make, which ends with the groff command. At this point, the execution of the %build section has been completed. Since the -bc option was used, RPM stops at this point.

The next step in the build process would be to install the newly built software. This is done in the %install section of the spec file. RPM can be stopped after the install has taken place by using the -bi option:

12.1.3 rpm -bi — Execute %prep, %build, %install

By using the `-bi` option, RPM is directed to stop once the software is completely built and installed on the build system. Here's what the output of a build using the `-bi` option looks like:

```
# rpm -bi cdplayer-1.0.spec
* Package: cdplayer
Executing: %prep
...
+ exit 0
Executing: %build
...
+ exit 0
Executing: %install
+ cd /usr/src/redhat/BUILD
+ cd cdplayer-1.0
+ make install
chmod 755 cdp
chmod 644 cdp.1.Z
cp cdp /usr/local/bin
ln -s /usr/local/bin/cdp /usr/local/bin/cdplay
cp cdp.1 /usr/local/man/man1
+ exit 0
+ umask 022
+ echo Executing: special doc
Executing: special doc
+ cd /usr/src/redhat/BUILD
+ cd cdplayer-1.0
+ DOCDIR=//usr/doc/cdplayer-1.0-1
+ rm -rf //usr/doc/cdplayer-1.0-1
+ mkdir -p //usr/doc/cdplayer-1.0-1
+ cp -ar README //usr/doc/cdplayer-1.0-1
+ exit 0
#
```

As before, we've excised most of the previously described sections. In this example, the `%install` section looks like:

```
%install
make install
```

After the `%prep` and `%build` sections, the `%install` section is executed. Looking at the output, we see that RPM changes directory into `cdplayer`'s top-level directory and issues the `make install` command, the sole command in the `%install` section. The output from that point until the first `exit 0`, is from `make install`.

The remaining commands are due to the contents of the spec file's `%files` list. Here's what it looks like:

```
%files
%doc README
/usr/local/bin/cdp
/usr/local/bin/cdplay
/usr/local/man/man1/cdp.1
```

The line responsible is `%doc README`. The `%doc` tag identifies the file as being documentation. RPM handles documentation files by creating a directory in `/usr/doc` and placing all documentation in it. The `exit 0` at the end signifies the end of the `%install` section. RPM stops due to the `-bi` option.

The next step at which RPM's build process can be stopped is after the software's binary package file has been created. This is done using the `-bb` option:

12.1.4 rpm -bb — Execute %prep, %build, %install, package (bin)

```
# rpm -bb cdplayer-1.0.spec
* Package: cdplayer
Executing: %prep
...
+ exit 0
Executing: %build
...
+ exit 0
Executing: %install
...
+ exit 0
Executing: special doc
...
+ exit 0
Binary Packaging: cdplayer-1.0-1
Finding dependencies...
Requires (2): libc.so.5 libncurses.so.2.0
usr/doc/cdplayer-1.0-1
usr/doc/cdplayer-1.0-1/README
usr/local/bin/cdp
usr/local/bin/cdplay
usr/local/man/man1/cdp.1
93 blocks
Generating signature: 0
Wrote: /usr/src/redhat/RPMS/i386/cdplayer-1.0-1.i386.rpm
+ umask 022
+ echo Executing: %clean
Executing: %clean
+ cd /usr/src/redhat/BUILD
+ cd cdplayer-1.0
+ exit 0
```

```
#
```

After executing the `%prep`, `%build`, and `%install` sections, and handling any special documentation files, RPM then creates a binary package file. In the sample output, we see that first RPM performs automatic dependency checking. It does this by determining which shared libraries are required by the executable programs contained in the package. Next, RPM actually archives the files to be packaged, optionally signs the package file, and outputs the finished product.

The last part of RPM's output looks suspiciously like a section in the spec file being executed. In our example, there is no `%clean` section. If there were, however, RPM would have executed any commands in the section. In the absence of a `%clean` section, RPM simply issues the usual `cd` commands and exits normally.

12.1.5 `rpm -ba` — Execute `%prep`, `%build`, `%install`, package (bin, src)

The `-ba` option directs RPM to perform *all* the stages in building a package. With this one command, RPM:

- Unpacks the original sources.
- Applies patches (if desired).
- Builds the software.
- Installs the software.
- Creates the binary package file.
- Creates the source package file.

That's quite a bit of work for one command! Here it is, in action:

```
# rpm -ba cdplayer-1.0.spec
* Package: cdplayer
Executing: %prep
...
+ exit 0
Executing: %build
...
+ exit 0
Executing: %install
...
+ exit 0
Executing: special doc
...
+ exit 0
Binary Packaging: cdplayer-1.0-1
```

```
...
Executing: %clean
...
+ exit 0
Source Packaging: cdplayer-1.0-1
cdplayer-1.0.spec
cdplayer-1.0.tgz
80 blocks
Generating signature: 0
Wrote: /usr/src/redhat/SRPMS/cdplayer-1.0-1.src.rpm
#
```

As in previous examples, RPM executes the `%prep`, `%build`, and `%install` sections, handles any special documentation files, creates a binary package file, and cleans up after itself.

The final step in the build process is to create a source package file. As the output shows, it consists of the spec file and the original sources. A source package may optionally include one or more patch files, although in our example, `cdplayer` requires none.

At the end of a build using the `-ba` option, the software has been successfully built and packaged in both binary and source form. But there are a few more build-time options that we can use. One of them is the `-bl` option:

12.1.6 `rpm -bl` — Check `%files` list

There's one last letter that may be specified with `rpm -b`, but unlike the others, which indicate the stage at which the build process is to stop, this option performs a variety of checks on the `%files` list in the named spec file. When `l` is added to `rpm -b`, the following checks are performed:

- Expands the spec file's `%files` list and checks that each file listed actually exists.
- Determines what shared libraries the software requires by examining every executable file listed.
- Determines what shared libraries are provided by the package.

Why is it necessary to do all this checking? When would it be useful? Keep in mind that the `%files` list must be generated manually. By using the `-bl` option, the following steps are all that's necessary to create a `%files` list:

- Writing the `%files` list.
- Using the `-bl` option to check the `%files` list.
- Making any necessary changes to the `%files` list.

It may take more than one iteration through these steps, but eventually the list check will pass. Using the `-bl` option to check the `%files` list is certainly better than starting a two-hour package build, only to find out at the very end that the list contains a misspelled filename.

Here's an example of the `-bl` option in action:

```
# rpm -bl cdplayer-1.0.spec
Package:  cdplayer
File List Check:  cdplayer-1.0-1
Finding dependencies...
Requires (2):  libc.so.5 libncurses.so.2.0
#
```

It's hard to see exactly what RPM is doing from the output, but if we add `-vv`, we can get a bit more information:

```
# rpm -bl -vv cdplayer-1.0.spec
D: Switched to BASE package
D: Source(0) = sunsite.unc.edu:/pub/Linux/apps/sound/cds/cdplayer-1.0.tgz
D: Switching to part: 12
D: fileFile =
D: Switched to package: (null)
D: Switching to part: 2
D: fileFile =
D: Switching to part: 3
D: fileFile =
D: Switching to part: 4
D: fileFile =
D: Switching to part: 10
D: fileFile =
D: Switched to package: (null)
* Package: cdplayer
File List Check: cdplayer-1.0-1
D: ADDING: /usr/doc/cdplayer-1.0-1
D: ADDING: /usr/doc/cdplayer-1.0-1/README
D: ADDING: /usr/local/bin/cdp
D: ADDING: /usr/local/bin/cdplay
D: ADDING: /usr/local/man/man1/cdp.1
D: md5(/usr/doc/cdplayer-1.0-1/README) = 2c149b2fb1a4d65418131a19b242601c
D: md5(/usr/local/bin/cdp) = 0f2a7a2f81812c75fd01c52f456798d6
D: md5(/usr/local/bin/cdplay) = d41d8cd98f00b204e9800998ecf8427e
D: md5(/usr/local/man/man1/cdp.1) = b32cc867ae50e2bdfa4d6780b084adfa
Finding dependencies...
D: Adding require: libncurses.so.2.0
D: Adding require: libc.so.5
Requires (2): libc.so.5 libncurses.so.2.0
#
```

Looking at this more verbose output, it's easy to see there's a great deal going on. Some of it is not directly pertinent to checking the `%files` list, however. For example, the output extending from the first line, to the line reading `* Package: cdplayer`, reflects processing that takes place during actual package building, and can be ignored.

Following that section is the actual `%files` list check. In this section, every file named in the `%files` list is checked to make sure it exists. The phrase, `ADDING:`, again reflects RPM's package building roots. When using the `-bl` option, however, RPM is simply making sure the files exist on the build system. If the `--timecheck` option (described a bit later, on page 159) is present, the checks required by that option are performed here, as well.

After the list check, the MD5 checksums of each file are calculated and displayed. While this information is vital during actual package building, it is not used when using the `-bl` option.

Finally, RPM determines which shared libraries the listed files require. In this case, there are only two — `libc.so.5`, and `libncurses.so.2.0`. While not strictly a part of the list-checking process, displaying shared library dependencies can be quite helpful at this point. It can point out possible problems, such as assuming that the target systems have a certain library installed when, in fact, they do not.

So far, we've only seen what happens when the `%files` list is correct. Let's see what happens where the list has problems. In this example, we've added a bogus file to the package's `%files` list:

```
# rpm -bl cdplayer-1.0.spec
Package:  cdplayer
File List Check:  cdplayer-1.0-1
File not found:  /usr/local/bin/bogus
Build failed.
#
```

Reflecting more of its package building roots, `rpm -bl` says that the "build failed". But the bottom line is that there is no such file as `/usr/bin/bogus`. In this example we made the name obviously wrong, but in a more real-world setting, the name will more likely be a misspelling in the `%files` list. OK, let's correct the `%files` list and try again:

```
# rpm -bl cdplayer-1.0.spec
Package:  cdplayer
File List Check:  cdplayer-1.0-1
File not found:  /usr/local/bin/cdplay
Build failed.
#
```

Another error! In this case the file is spelled correctly, but it is not on the build system, even though it should be. Perhaps it was deleted accidentally. In any case, let's rebuild the software and try again:

```
# rpm -bi cdplayer-1.0.spec
* Package: cdplayer
Executing: %prep
...
+ exit 0
Executing: %build
...
+ exit 0
Executing: %install
...
ln -s /usr/local/bin/cdp /usr/local/bin/cdplay
...
+ exit 0
Executing: special doc
...
+ exit 0
#
```

```
# rpm -bl cdplayer-1.0.spec
Package:  cdplayer
File List Check:  cdplayer-1.0-1
Finding dependencies...
Requires (2):  libc.so.5 libncurses.so.2.0
#
```

Done! The moral to this story is that using `rpm -bl` and fixing the error it flagged doesn't necessarily mean your `%files` list is ready for prime-time: Always run it again to make sure!

12.1.7 `--short-circuit` — Force build to start at particular stage

Although it sounds dangerous, the `--short-circuit` option can be your friend. This option is used during the initial development of a package. Earlier in the chapter, we explored stopping RPM's build process at different stages. Using `--short-circuit`, we can *start* the build process at different stages.

One time that `--short-circuit` comes in handy is when you're trying to get software to build properly. Just think what it would be like — you're hacking away at the sources, trying a build, getting an error, and hacking some more to fix that error. Without `--short-circuit`, you'd have to:

1. Make your change to the sources.
2. Use `tar` to create a new source archive.
3. Start a build with something like `rpm -bc`.

4. See another bug.

5. Go back to step 1.

Pretty cumbersome! Since RPM's build process is designed to start with the sources in their original `tar` file, unless your modifications end up in that `tar` file, they won't be used in the next build.[1]

But there's another way. Just follow these steps:

1. Place the original source `tar` file in RPM's `SOURCES` directory.

2. Create a partial spec file in RPM's `SPECS` directory (Be sure to include a valid `Source` line).

3. Issue an `rpm -bp` to properly create the build environment.

Now use `--short-circuit` to attempt a compile. Here's an example:

```
# rpm -bc --short-circuit cdplayer-1.0.spec
* Package: cdplayer
+ umask 022
+ echo Executing: %build
Executing: %build
+ cd /usr/src/redhat/BUILD
+ cd cdplayer-1.0
+ make
gcc -Wall -O2  -c -I/usr/include/ncurses  cdp.c
gcc -Wall -O2  -c -I/usr/include/ncurses  color.c
gcc -Wall -O2  -c -I/usr/include/ncurses  display.c
gcc -Wall -O2  -c -I/usr/include/ncurses  misc.c
gcc -Wall -O2  -c -I/usr/include/ncurses  volume.c
volume.c: In function ‘mix_set_volume’:
volume.c:67: warning: implicit declaration of function ‘ioctl’
gcc -Wall -O2  -c -I/usr/include/ncurses  hardware.c
gcc -Wall -O2  -c -I/usr/include/ncurses  database.c
gcc -Wall -O2  -c -I/usr/include/ncurses  getline.c
gcc -o cdp cdp.o color.o display.o misc.o volume.o
     hardware.o database.o getline.o  -I/usr/include/ncurses
     -L/usr/lib -lncurses
groff -Tascii -man cdp.1 | compress >cdp.1.Z
+ exit 0
#
```

[1] As we mentioned in chapter 10, if the original sources need to be modified, the modifications should be kept as a separate set of patches. However, during development, it makes more sense to not generate patches every time a change to the original source is made.

Normally, the `-bc` option instructs RPM to *stop* the build after the `%build` section of the spec file has been executed. By adding `--short-circuit`, however, RPM *starts* the build by executing the `%build` section and stops when everything in `%build` has been executed.

There is only one other build stage that can be `--short-circuit`'ed, and that is the install stage. The reason for this restriction is to make it difficult to bypass RPM's use of pristine sources. If it were possible to `--short-circuit` to `-bb` or `-ba`, a package builder might take the "easy" way out and simply hack at the build tree until the software built successfully, then package the hacked sources. So, RPM will only `--short-circuit` to `-bc` or `-bi`. Nothing else will do.

What exactly does an `rpm -bi --short-circuit` do, anyway? Like an `rpm -bc --short-circuit`, it starts executing at the named stage, which in this case is `%install`. Note that the build environment must be ready to perform an install before attempting to `--short-circuit` to the `%install` stage. If the software installs via `make install`, `make` will automatically compile the software anyway.

And what happens if the build environment isn't ready and a `--short-circuit` is attempted? Let's see:

```
# rpm -bi --short-circuit cdplayer-1.0.spec
* Package: cdplayer
+ umask 022
+ echo Executing: %install
Executing: %install
+ cd /usr/src/redhat/BUILD
+ cd cdplayer-1.0
/var/tmp/rpmbu01157aaa: cdplayer-1.0: No such file or directory
Bad exit status
#
```

RPM blindly started executing the `%install` stage, but came to an abrupt halt when it attempted to change directory into `cdplayer-1.0`, which didn't exist. After giving a descriptive error message, RPM exited with a failure status. Except for some minor differences, `rpm -bc` would have failed in the same way.

12.1.8 --buildarch *<arch>* — Perform Build For the *<arch>* Architecture

The `--buildarch` option is used to override RPM's architecture detection logic. The option is followed by the desired architecture name. Here's an example:

```
# rpm -ba --buildarch i486 cdplayer-1.0.spec
Package:  cdplayer
...
Binary Packaging:  cdplayer-1.0-1
...
```

```
Wrote: /usr/src/redhat/RPMS/i486/cdplayer-1.0-1.i486.rpm
...
Wrote: /usr/src/redhat/SRPMS/cdplayer-1.0-1.src.rpm
#
```

We've removed most of RPM's output from this example, but the main thing we can see from this example is that the package was built for the `i486` architecture, due to the inclusion of the `--buildarch` option on the command line. We can also see that RPM wrote the binary package in the architecture-specific directory, `/usr/src/redhat/RPMS/i486`. Using RPM's `--queryformat` option confirms the package's architecture:

```
# rpm -qp --queryformat '%{arch}\n'\
> /usr/src/redhat/RPMS/i486/cdplayer-1.0-1.i486.rpm
i486
#
```

For more information on build packages for multiple architectures, please see Chapter 19 on page 269.

12.1.9 --buildos *<os>* — Perform Build For the *<os>* Operating System

The `--buildos` option is used to override RPM's operating system detection logic. The option is followed by the desired operating system name. Here's an example:

```
# rpm -ba --buildos osf1 cdplayer-1.0.spec
...
Binary Packaging: cdplayer-1.0-1
...
Wrote: /usr/src/redhat/RPMS/i386/cdplayer-1.0-1.i386.rpm
Source Packaging: cdplayer-1.0-1
...
Wrote: /usr/src/redhat/SRPMS/cdplayer-1.0-1.src.rpm
#
```

There's nothing in the build output that explicitly states the build operating system as been set to `osf1`. Let's see if `--queryformat` will tell us:

```
# rpm -qp --queryformat '%{os}\n'\
> /usr/src/redhat/RPMS/i386/cdplayer-1.0-1.i386.rpm
osf1
#
```

The package was indeed built for the specified operating system. For more information on building packages for multiple operating systems, please see Chapter 19 on page 269.

12.1.10 --sign — Add a Digital Signature to the Package

The --sign option directs RPM to add a digital signature to the package being built. Currently, this is done using PGP. Here's an example of --sign in action:

```
# rpm -ba --sign cdplayer-1.0.spec
Enter pass phrase: <passphrase> (not echoed)
Pass phrase is good.
Package: cdplayer
...
Binary Packaging: cdplayer-1.0-1
...
Generating signature: 1002
Wrote: /usr/src/redhat/RPMS/i386/cdplayer-1.0-1.i386.rpm
...
Source Packaging: cdplayer-1.0-1
...
Generating signature: 1002
Wrote: /usr/src/redhat/SRPMS/cdplayer-1.0-1.src.rpm
#
```

The most obvious effect of adding the --sign option to a build command is that RPM then asks for your private key's passphrase. After entering the passphrase (which isn't echoed), the build proceeds as usual. The only other difference between this and a non-signed build is that the `Generating signature:` lines have a non-zero value.

Let's check the source and binary packages we've just created and see if they are, in fact, signed:

```
# rpm --checksig /usr/src/redhat/SRPMS/cdplayer-1.0-1.src.rpm
/usr/src/redhat/SRPMS/cdplayer-1.0-1.src.rpm: size pgp md5 OK
# rpm --checksig /usr/src/redhat/RPMS/i386/cdplayer-1.0-1.i386.rpm
/usr/src/redhat/RPMS/i386/cdplayer-1.0-1.i386.rpm: size pgp md5 OK
#
```

The fact that there is a pgp in --checksig's output indicates that the packages have been signed.

For more information on signing packages, please see Chapter 17 on page 241. Appendix G on page 427 contains information on obtaining and installing PGP.

12.1.11 --test — Create, Save Build Scripts For Review

There are times when it might be necessary to get a more in-depth view of a particular build. By using the `--test` option, it's easy. When `--test` is added to a build command, the scripts RPM would normally use to actually perform the build, are created and saved for you to review. Let's see how it works:

```
# rpm -ba --test cdplayer-1.0.spec
Package:  cdplayer
#
```

Unlike a normal build, there's not much output. But the `--test` option has caused a set of scripts to be written and saved for you. The question is: Where are they?

If you are using a customized rpmrc file, the scripts will be written to the directory specified by the rpmrc entry `tmppath`. If you haven't changed this setting, RPM, by default, writes the scripts in `/var/tmp`. Here they are:

```
# ls -l /var/tmp
total 4
-rw-rw-r--   1 root     root          670 Sep 17 20:35 rpmbu00236aaa
-rw-rw-r--   1 root     root          449 Sep 17 20:35 rpmbu00236baa
-rw-rw-r--   1 root     root          482 Sep 17 20:35 rpmbu00236caa
-rw-rw-r--   1 root     root          552 Sep 17 20:35 rpmbu00236daa
#
```

Each file contains a script that performs a given part of the build. Here's the first file:

```
#!/bin/sh -e
# Script generated by rpm

RPM_SOURCE_DIR="/usr/src/redhat/SOURCES"
RPM_BUILD_DIR="/usr/src/redhat/BUILD"
RPM_DOC_DIR="/usr/doc"
RPM_OPT_FLAGS="-O2 -m486 -fno-strength-reduce"
RPM_ARCH="i386"
RPM_OS="Linux"
RPM_ROOT_DIR="/tmp/cdplayer"
RPM_BUILD_ROOT="/tmp/cdplayer"
RPM_PACKAGE_NAME="cdplayer"
RPM_PACKAGE_VERSION="1.0"
RPM_PACKAGE_RELEASE="1"
set -x

umask 022
```

```
echo Executing: %prep
cd /usr/src/redhat/BUILD

cd /usr/src/redhat/BUILD
rm -rf cdplayer-1.0
gzip -dc /usr/src/redhat/SOURCES/cdplayer-1.0.tgz | tar -xvvf -
if [ $? -ne 0 ]; then
  exit $?
fi
cd cdplayer-1.0
cd /usr/src/redhat/BUILD/cdplayer-1.0
chown -R root.root .
chmod -R a+rX,g-w,o-w .
```

As we can see, this script contains the `%prep` section from the spec file. The script starts off by defining a number of environment variables and then leads into the `%prep` section. In the spec file used in this build, the `%prep` section consists of a single `%setup` macro. In this file, we can see exactly how RPM expands that macro. The remaining files follow the same basic layout — a section defining environment variables, followed by the commands to be executed.

Note that the `--test` option will only create script files for each build stage, as specified in the command line. For example, if the above command was changed to:

```
# rpm -bp --test cdplayer-1.0.spec
#
```

only one script file, containing the `%prep` commands, would be written. In any case, no matter what RPM build command is used, the `--test` option can let you see exactly what is going to happen during a build.

12.1.12 `--clean` — Clean up after build

The `--clean` option can be used to ensure that the package's build directory tree is removed at the end of a build. Although it can be used with any build stage, it doesn't always make much sense to do so:

```
# rpm -bp --clean cdplayer-1.0.spec
* Package: cdplayer
Executing: %prep
...
+ exit 0
+ echo Executing: sweep
Executing: sweep
+ cd /usr/src/redhat/BUILD
+ rm -rf cdplayer-1.0
```

```
+ exit 0
#
```

In this example, we see a typical `%prep` section being executed. The line "`+ echo Executing: sweep`" indicates the start of `--clean`'s activity. After changing directory into the build directory, RPM then issues a recursive delete on the package's top-level directory.

As we noted above, this particular example doesn't make much sense. We're only executing the `%prep` section, which creates the package's build tree, and using `--clean`, which removes it! Using `--clean` with the `-bc` option isn't very productive either, as the newly built software remains in the build tree. Once again, there would be no remnants left after `--clean` has done its thing.

Normally, the `--clean` option is used once the software builds and can be packaged successfully. It is particularly useful when more than one package is to be built, since `--clean` ensures that the filesystem holding the build area will not fill up with build trees from each package.

Note also that the `--clean` option only removes the files that reside in the software's build tree. If there are any files that the build creates outside of this hierarchy, it will be necessary to write a script for the spec file's `%clean` section.

12.1.13 `--buildroot` *<path>* — Execute `%install` using *<path>* as the root

The `--buildroot` option can make two difficult situations much easier:

- Performing a build without impacting the build system.
- Allowing non-root users to build packages.

Let's study the first situation in a bit more detail. Say, for example, that `sendmail` is to be packaged. In the course of creating a `sendmail` package, the software must be installed. This would mean that critical `sendmail` files, such as `sendmail.cf` and aliases, would be overwritten. Mail handling on the build system would almost certainly be disrupted.

In the second case, it's certainly possible to set permissions such that non-root users can install software, but highly unlikely that any system administrator worth their salt would do so. What can be done to make these situations more tenable?

The `--buildroot` option is used to instruct RPM to use a directory other than `/` as a "build root". This phrase is a bit misleading, in that the build root is *not* the root directory under which the software is built. Rather, it is the root directory for the install phase of the build. When a build root is not specified, the software being packaged is installed relative to the build system's root directory "`/`".

However, it's not enough to just specify a build root on the command line. The spec file for the package must be set up to support a build root. If you don't make the necessary changes, this is what you'll see:

```
# rpm -ba --buildroot /tmp/foo cdplayer-1.0.spec
Package can not do build prefixes
Build failed.
#
```

Chapter 16 has complete instructions on the modifications necessary to configure a package to use an alternate build root, as well as methods to permit users to build packages without root access. Assuming that the necessary modifications have been made, here is what the build would look like:

```
# rpm -ba --buildroot /tmp/foonly cdplayer-1.0.spec
* Package: cdplayer
Executing: %prep
+ cd /usr/src/redhat/BUILD
...
+ exit 0
Executing: %build
+ cd /usr/src/redhat/BUILD
+ cd cdplayer-1.0
...
+ exit 0
+ umask 022
Executing: %install
+ cd /usr/src/redhat/BUILD
+ cd cdplayer-1.0
+ make ROOT=/tmp/foonly install
install -m 755 -o 0 -g 0 -d /tmp/foonly/usr/local/bin/
install -m 755 -o 0 -g 0 cdp /tmp/foonly/usr/local/bin/cdp
rm -f /tmp/foonly/usr/local/bin/cdplay
ln -s /tmp/foonly/usr/local/bin/cdp /tmp/foonly/usr/local/bin/cdplay
install -m 755 -o 0 -g 0 -d /tmp/foonly/usr/local/man/man1/
install -m 755 -o 0 -g 0 cdp.1 /tmp/foonly/usr/local/man/man1/cdp.1
+ exit 0
Executing: special doc
+ cd /usr/src/redhat/BUILD
+ cd cdplayer-1.0
+ DOCDIR=/tmp/foonly//usr/doc/cdplayer-1.0-1
+ rm -rf /tmp/foonly//usr/doc/cdplayer-1.0-1
+ mkdir -p /tmp/foonly//usr/doc/cdplayer-1.0-1
+ cp -ar README /tmp/foonly//usr/doc/cdplayer-1.0-1
+ exit 0
Binary Packaging: cdplayer-1.0-1
Finding dependencies...
Requires (2): libc.so.5 libncurses.so.2.0
usr/doc/cdplayer-1.0-1
```

```
usr/doc/cdplayer-1.0-1/README
usr/local/bin/cdp
usr/local/bin/cdplay
usr/local/man/man1/cdp.1
93 blocks
Generating signature: 0
Wrote: /usr/src/redhat/RPMS/i386/cdplayer-1.0-1.i386.rpm
+ umask 022
+ echo Executing: %clean
Executing: %clean
+ cd /usr/src/redhat/BUILD
+ cd cdplayer-1.0
+ exit 0
Source Packaging: cdplayer-1.0-1
cdplayer-1.0.spec
cdplayer-1.0.tgz
82 blocks
Generating signature: 0
Wrote: /usr/src/redhat/SRPMS/cdplayer-1.0-1.src.rpm
#
```

As the somewhat edited output shows, the `%prep`, `%build`, and `%install` sections are executed in RPM's normal build directory. However, the `--buildroot` option comes into play when the `make install` is done. As we can see, the `ROOT` variable is set to `/tmp/foonly`, which was the value following `--buildroot` on the command line. From that point on, we can see that `make` substituted the new build root value during the install phase.

The build root is also used when documentation files are installed. The documentation directory `cdplayer-1.0-1` is created in `/tmp/foonly/usr/doc`, and the README file is placed in it.

The only remaining difference that results from using `--buildroot`, is that the files to be included in the binary package are not located relative to the build system's root directory. Instead they are located relative to the build root `/tmp/foonly`. The resulting binary and source package files are functionally equivalent to packages built without the use of `--buildroot`.

Using `--buildroot` Can Bite You!

Although the `--buildroot` option can solve some problems, using a build root can actually be dangerous. How? Consider the following situation:

- A spec file is configured to have a build root of `/tmp/blather`, for instance.
- In the `%prep` section[2], there is an `rm -rf $RPM_BUILD_ROOT` command to clean out any old installed software.

[2] Or the `%clean` section, it doesn't matter — the end result is the same.

- You decide to build the software so that it installs relative to your system's root directory, so you enter the following command: "`rpm -ba --buildroot / foo.spec`".

The end result? Since specifying "/" as the build root sets $RPM_BUILD_ROOT to "/", that innocuous little `rm -rf $RPM_BUILD_ROOT` turns into `rm -rf /`! A recursive delete, starting at your system's root directory, might not be a total disaster if you catch it quickly, but in either case, you'll be testing your ability to restore from backup... Er, you *do* have backups, don't you?

The moral of this story is to be *very* careful when using `--buildroot`. A good rule of thumb is to always specify a unique build root. For example, instead of specifying `/tmp` as a build root (and possibly losing your system's directory for holding temporary files), use the path `/tmp/mypackage`, where the directory `mypackage` is used only by the package you're building.

12.1.14 `--timecheck` *<secs>* — Print a warning if files to be packaged are over *<secs>* old

While it's possible to detect many errors in the `%files` list using `rpm -bl`, there is another type of problem that can't be detected. Consider the following scenario:

- A package you're building creates the file `/usr/bin/foo`.
- Because of a problem with the package's makefile, `foo` is never copied into `/usr/bin`.
- An older, incompatible version of `foo`, created several months ago, already exists in `/usr/bin`.
- RPM creates the binary package file.

Is the incompatible `/usr/bin/foo` included in the package? You bet it is! If only there was some way for RPM to catch this type of problem...

Well, there is! By adding `--timecheck`, followed by a number, RPM will check each file being packaged, to see if the file is more than the specified number of seconds old. If it is, a warning message is displayed. The `--timecheck` option works with either the `-ba` or `-bl` options. Here's an example using `-bl`:

```
# rpm -bl --timecheck 3600 cdplayer-1.0.spec
* Package: cdplayer
File List Check: cdplayer-1.0-1
warning: TIMECHECK failure: /usr/doc/cdplayer-1.0-1/README
Finding dependencies...
Requires (2): libc.so.5 libncurses.so.2.0
#
```

In this example, the file `/usr/doc/cdplayer-1.0-1/README` is more than 3,600 seconds, or one hour, old. If we take a look at the file, we find that it is:[3]

[3] It should be noted that the package was built *substantially* later than November of 1995!

```
# ls -al /usr/doc/cdplayer-1.0-1/README
-rw-r--r-- 1 root root 1085 Nov 10 1995 README
#
```

In this particular case, the warning from `--timecheck` is no cause for alarm. Since the `README` file was simply copied from the original source, which was created November 10th, 1995, its date is unchanged. If the file had been an executable or a library that was supposedly built recently, `--timecheck`'s warning should be taken more seriously.

If you'd like to set a default time check value of one hour, you can include the following line in your `rpmrc` file:

```
timecheck: 3600
```

This value can still be overridden by a value on the command line, if desired. For more information on the use of `rpmrc` files, see Appendix B.

12.1.15 `-vv` — Display debugging information

Unlike most other RPM commands, there is no `-v` option for `rpm -b`. That's because the command's default is to be verbose. However, even more information can be obtained by adding `-vv`. Here's an example:

```
# rpm -bp -vv cdplayer-1.0.spec
D: Switched to BASE package
D: Source(0) = sunsite.unc.edu:/pub/Linux/apps/sound/cds/cdplayer-1.0.tgz
D: Switching to part: 12
D: fileFile =
D: Switched to package: (null)
D: Switching to part: 2
D: fileFile =
D: Switching to part: 3
D: fileFile =
D: Switching to part: 4
D: fileFile =
D: Switching to part: 10
D: fileFile =
D: Switched to package: (null)
* Package: cdplayer
D: RUNNING: %prep
+ umask 022
+ echo Executing: %prep
Executing: %prep
+ cd /usr/src/redhat/BUILD
```

```
+ cd /usr/src/redhat/BUILD
+ rm -rf cdplayer-1.0
+ gzip -dc /usr/src/redhat/SOURCES/cdplayer-1.0.tgz
+ tar -xvvf -
drwxrwxr-x root/users        0 Aug  4 22:30 1996 cdplayer-1.0/
-rw-r--r-- root/users    17982 Nov 10 01:10 1995 cdplayer-1.0/COPYING
...
-rw-r--r-- root/users     1515 Nov 10 01:10 1995 cdplayer-1.0/volume.h
+ [ 0 -ne 0 ]
+ cd cdplayer-1.0
+ cd /usr/src/redhat/BUILD/cdplayer-1.0
+ chown -R root.root .
+ chmod -R a+rX,g-w,o-w .
+ exit 0
#
```

Most of the output generated by the `-vv` option is preceded by a `D:`. In this example, the additional output represents RPM's internal processing during the start of the build process. Using the `-vv` option with other build commands will produce different output.

12.1.16 `--quiet` — Produce as Little Output as Possible

As we mentioned above, the build command is normally verbose. The `--quiet` option can be used to cut down on the command's output:

```
# rpm -ba --quiet cdplayer-1.0.spec
Package:  cdplayer
volume.c:  In function 'mix_set_volume':
volume.c:67:  warning:  implicit declaration of function 'ioctl'
90 blocks
82 blocks
#
```

This is the entire output from a package build of `cdplayer`. Note that warning messages (actually, anything sent to stdout) are still printed.

12.1.17 `--rcfile` *<rcfile>* — Set alternate rpmrc file to *<rcfile>*

The `--rcfile` option is used to specify a file containing default settings for RPM. Normally, this option is not needed. By default, RPM uses `/etc/rpmrc` and a file named `.rpmrc` located in your login directory.

This option would be used if there was a need to switch between several sets of RPM defaults. Software developers and package builders will normally be the only people using the `--rcfile` option. For more information on `rpmrc` files, see Appendix B.

12.2 Other Build-related Commands

There are two other commands that also perform build-related functions. However, they do not use the `rpm -b` command syntax that we've been studying so far. Instead of specifying the name of the spec file, as with `rpm -b`, it's necessary to specify the name of the source package file.

Why the difference in syntax? The reason has to do with the differing functions of these commands. Unlike `rpm -b`, where the name of the game is to get software packaged into binary and source package files, these commands use an already-existing source package file as input. Let's take a look at them:

12.2.1 `rpm --recompile` — What Does it Do?

The `--recompile` option directs RPM to perform the following steps:

- Install the specified source package file.
- Unpack the original sources.
- Build the software.
- Install the software.
- Remove the software's build directory structure.

While you might think this sounds a great deal like an install of the source package file, followed by an `rpm -bi`, this is not entirely the case. Using `--recompile`, the only file required is the source package file. After the software is built and installed, the only thing left, other than the newly installed software, is the original source package file.

The `--recompile` option is normally used when a previously installed package needs to be recompiled. `--recompile` comes in handy when software needs to be compiled against a new version of the kernel.

Here's what RPM displays during a `--recompile`:

```
# rpm --recompile cdplayer-1.0-1.src.rpm
Installing cdplayer-1.0-1.src.rpm
* Package: cdplayer
Executing: %prep
...
+ exit 0
Executing: %build
...
+ exit 0
Executing: %install
...
+ exit 0
```

```
Executing: special doc
...
+ exit 0
Executing: sweep
...
+ exit 0
#
```

The very first line shows RPM installing the source package. After that are ordinary executions of the `%prep`, `%build`, and `%install` sections of the spec file. Finally, the cleanup of the software's build directory takes place, just as if the `--clean` option had been specified.

Since `rpm -i` or `rpm -U` are not being used to install the software, the RPM database is not updated during a `--recompile`. This means that doing a `--recompile` on an already-installed package may result in problems down the road, when RPM is used to upgrade or verify the package.

12.2.2 `rpm --rebuild` — What Does it Do?

Package builders, particularly those that create packages for multiple architectures, often need to build their packages starting from the original sources. The `--rebuild` option does this, starting from a source package file. Here is the list of steps it performs:

- Install the specified source package file.
- Unpack the original sources.
- Build the software.
- Install the software.
- Create a binary package file.
- Remove the software's build directory tree.

Like the `--recompile` option, `--rebuild` cleans up after itself. The only difference between the two commands is the fact that `--rebuild` also creates a binary package file. The only remnants of a `--rebuild` are the original source package, the newly installed software, and a new binary package file.

Package builders find this command especially handy, as it allows them to create new binary packages using one command, with no additional cleanups required. There are several times when `--rebuild` is normally used:

- When the build environment (eg. compilers, libraries, etc.) has changed.
- When binary packages for a different architecture are to be built.

Here's an example of the `--rebuild` option in action:

```
# rpm --rebuild cdplayer-1.0-1.src.rpm
```

```
Installing cdplayer-1.0-1.src.rpm
* Package: cdplayer
Executing: %prep
...
+ exit 0
Executing: %build
...
+ exit 0
Executing: %install
...
+ exit 0
Executing: special doc
...
+ exit 0
Binary Packaging: cdplayer-1.0-1
...
Executing: %clean
...
+ exit 0
Executing: sweep
...
+ exit 0
#
```

The very first line shows RPM installing the source package. The lines after that are ordinary executions of the `%prep`, `%build`, and `%install` sections of the spec file. Next, a binary package file is created. Finally, the spec file's `%clean` section (if one exists) is executed. The cleanup of the software's build directory takes place, just as if the `--clean` option had been specified.

That completes our overview of the commands used to build packages with RPM. In the next chapter, we'll look at the various macros that are available and how they can make life easier for the package builder.

Chapter 13

Inside the Spec File

In this chapter, we're going to cover the spec file in detail. There are a number of different types of entries that comprise a spec file, and every one will be documented here. The different types of entries are:

- Comments — Human-readable notes ignored by RPM.
- Tags — Define data.
- Scripts — Contain commands to be executed at specific times.
- Macros — A method of executing multiple commands easily.
- The `%files` list — A list of files to be included in the package.
- Directives — Used in the `%files` list to direct RPM to handle certain files in a specific way.
- Conditionals — Permit operating system- or architecture-specific preprocessing of the spec file.

Let's start by looking at comments.

13.1 Comments: Notes Ignored by RPM

Comments are a way to make RPM ignore a line in the spec file. The contents of a comment line are entirely up to the person writing the spec file.

To create a comment, enter an octothorp (`#`) at the start of the line. Any text following the comment character will be ignored by RPM. Here's an example comment:

```
# This is the spec file for playmidi 2.3...
```

Comments can be placed in any section of the spec file.

13.2 Tags: Data Definitions

Looking at a spec file, the first thing you'll see are a number of lines, all following the same basic format:

```
<something>:<something-else>
```

The `<something>` is known as a "tag", because it is used by RPM to name or *tag* some data. The tag is separated from its associated data by a colon. The data is represented by the `<something-else>` above. Tags are grouped together at the top of the spec file, in a section known as the preamble. Here's an example of a tag and its data:

```
Vendor:  White Socks Software, Inc.
```

In this example, the tag is "`Vendor`". Tags are not case-sensitive — they may be all uppercase, all lowercase, or anything in-between. The `Vendor` tag is used to define the name of the organization producing the package. The data in this example is "`White Socks Software, Inc.`". Therefore, RPM will store `White Socks Software, Inc.` as the vendor of the package.

Note, also, that spacing between the tag, the colon, and the data is unimportant. Given this, and the case-insensitivity of the tag, each of the following lines are equivalent to the one above:

```
VeNdOr :  White Socks Software, Inc.

vendor:White Socks Software, Inc.

VENDOR    :    White Socks Software, Inc.
```

The bottom line is that you can make tag lines as neat or as ugly as you like – RPM won't mind either way. Note, however, the tag's data may need to be formatted in a particular fashion. If there are any such restrictions, we'll mention them. Below, we've grouped tags of similar functions together for easier reference, starting with the tags that are used to create the package name.

13.2.1 Package Naming Tags

The following tags are used by RPM to produce the package's final name. Since the name is always in the format:

```
<name>-<version>-<release>
```

it's only natural that the three tags are known as `name`, `version`, and `release`.

The `name` Tag

The `name` tag is used to define the name of the software being packaged. In most (if not all) cases, the name used for a package should be identical in spelling and case to the software being packaged. The name cannot contain any whitespace: If it does, RPM will only use the first part of the name (up to the first space). Therefore, if the name of the software being packaged is `cdplayer`, the `name` tag should be something like:

```
Name: cdplayer
```

The `version` Tag

The `version` tag defines the version of the software being packaged. The version specified should be as close as possible to the format of the original software's version. In most cases, there should be no problem specifying the version just as the software's original developer did. However, there is a restriction. There can be no dashes in the version. If you forget, RPM will remind you:

```
# rpm -ba cdplayer-1.0.spec
Package: cdplayer
Illegal '-' char in version: 1.0-a
#
```

Spaces in the version will also cause problems, in that anything after the first space will be ignored by RPM. Bottom line: Stick with alphanumeric characters and periods, and you'll never have to worry about it. Here's a sample `version` tag:

```
Version: 1.2
```

The `release` Tag

The `release` tag can be thought of as the *package's* version. The release is traditionally an integer — for example, when a specific piece of software at a particular version is first packaged, the release should be "1". If it is necessary to repackage that software at the same version, the release should be incremented. When a new version of the software becomes available, the release should drop back to "1" when it is first packaged.

Note that we used the word "traditionally", above. The only hard and fast restriction to the release format is that there can be no dashes in it. Be aware that if you buck tradition, your users may not understand what your release means.

It is up to the package builder to determine which build represents a new release and to update the release manually. Here is what a typical `release` tag might look like:

```
Release: 5
```

13.2.2 Descriptive Tags

These tags provide information primarily for people who want to know a bit more about the package, and who produced it. They are part of the package file, and most of them can be seen by issuing an `rpm -qi` command.

The `%description` Tag

The `%description` tag is used to provide an in-depth description of the packaged software. The description should be several sentences describing, to an uninformed user, what the software does.

The `%description` tag is a bit different than the other tags in the preamble. For one, it starts with a percent sign. The other difference is that the data specified by the `%description` tag can span more than one line. In addition, a primitive formatting capability exists. If a line starts with a space, that line will be displayed verbatim by RPM. Lines that do not start with a space are assumed to be part of a paragraph and will be formatted by RPM. It's even possible to mix and match formatted and unformatted lines. Here are some examples:

```
%description
It slices!  It dices!  It's a CD player app that can't be beat.  By using
the resonant frequency of the CD itself, it is able to simulate 20X
oversampling.  This leads to sound quality that cannot be equaled with
more mundane software...
```

The example above contains no explicit formatting. RPM will format the text as a single paragraph, breaking lines as needed.

```
%description
 It slices!
 It dices!
 It's a CD player app that can't be beat.
By using the resonant frequency of the CD itself, it is able to simulate
20X oversampling.  This leads to sound quality that cannot be equaled with
more mundane software...
```

In this example, the first three lines will be displayed by RPM, verbatim. The remainder of the text will be formatted by RPM. The text will be formatted as one paragraph.

```
%description
 It slices!
 It dices!
 It's a CD player app that can't be beat.

By using the resonant frequency of the CD itself, it is able to simulate
20X oversampling.  This leads to sound quality that cannot be equaled with
more mundane software...
```

Above, we have a similar situation to the previous example, in that part of the text is formatted and part is not. However, the blank line separates the text into two paragraphs.

The `summary` Tag

The `summary` tag is used to define a one-line description of the packaged software. Unlike `%description`, `summary` is restricted to one line. RPM uses it when a succinct description of the package is needed. Here is an example of a `summary` line:

```
Summary:  A CD player app that rocks!
```

The `copyright` Tag

The `copyright` tag is used to define the copyright terms applicable to the software being packaged. In many cases, this might be nothing more than "`GPL`", for software distributed under the terms of the GNU General Public License, or something similar. For example:

```
Copyright:  GPL
```

The `distribution` Tag

The `distribution` tag is used to define a group of packages, of which this package is a part. Since Red Hat Software is in the business of producing a group of packages known as a Linux *distribution*, the name stuck. For example, if a suite of applications known as "Doors '95" were produced, each package that is part of the suite would define its `distribution` line like this:

```
Distribution:  Doors '95
```

The `icon` Tag

The `icon` tag is used to name a file containing an icon representing the packaged software. The file may be in either GIF or XPM format, although XPM is preferred. In either case, the background of the icon should be transparent. The file should be placed in RPM's `SOURCES` directory prior to performing a build, so no path is needed.

The icon is normally used by graphically-oriented front ends to RPM. RPM itself doesn't use the icon, but it's stored in the package file and retained in RPM's database after the package is installed. An example `icon` tag might look like:

```
Icon:  foo.xpm
```

The `vendor` Tag

The `vendor` tag is used to define the name of the entity that is responsible for packaging the software. Normally, this would be the name of an organization. Here's an example:

```
Vendor:  White Socks Software, Inc.
```

The `url` Tag

The `url` tag is used to define a Uniform Resource Locator that can be used to obtain additional information about the packaged software. At present, RPM doesn't actively make use of this tag. The data is stored in the package however, and will be written into RPM's database when the package is installed. It's only a matter of time before some web-based RPM adjunct makes use of this information, so make sure you include URLs! Something like this is all you'll need:

```
URL: http://www.gnomovision.com/cdplayer.html
```

The `group` Tag

The `group` tag is used to group packages together by the types of functionality they provide. The group specification looks like a path and is similar in function, in that it specifies more general groupings before more detailed ones. For example, a package containing a text editor might have the following group:

```
Group: Applications/Editors
```

In this example, the package is part of the `Editors` group, which is itself a part of the `Applications` group. Likewise, a spreadsheet package might have this group:

```
Group: Applications/Spreadsheets
```

This `group` tag indicates that under the `Applications` group, we would find `Editors` and `Spreadsheets`, and probably some other subgroups as well.

How is this information used? It's primarily meant to permit graphical front-ends to RPM, to display packages in a hierarchical fashion. Of course, in order for groups to be as effective as possible, it's necessary for all package builders to be consistent in their groupings. In the case of packages for Linux, Red Hat Software has the definitive list. Therefore, Linux package builders should give serious consideration to using Red Hat Software's groups. The current group hierarchy is installed with every copy of RPM, and is available in the RPM sources as well. Check out the file `groups` in RPM's documentation directory (normally `/usr/doc/rpm-`*<version>*), or in the top-level source directory.

The `packager` Tag

The `packager` tag is used to hold the name and contact information for the person or persons who built the package. Normally, this would be the person that actually built the package, or in a larger organization, a public relations contact. In either case, contact information such as an e-mail address or phone number should be included, so customers can send either money or hate mail, depending on their satisfaction with the packaged software. Here's an example of a `packager` tag:

```
Packager:  Fred Foonly <fred@gnomovision.com>
```

13.2.3 Dependency Tags

One RPM feature that's been recently implemented is a means of ensuring that if a package is installed, the system environment has everything the package requires in order to operate properly. Likewise, when an installed package is erased RPM can make sure no other package relies on the package being erased. This dependency capability can be very helpful when endusers install and erase packages on their own. It makes it more difficult for them to paint themselves into a corner, package-wise.

However, in order for RPM to be able to take more than basic dependency information into account, the package builder must add the appropriate dependency information to the package. This is done by using the following tags. Note, however, that adding dependency information to a package requires some forethought. For additional information on RPM's dependency processing, please review chapter 14 on page 209.

The `provides` Tag

The `provides` tag is used to specify a "virtual package" that the packaged software makes available when it is installed. Normally, this tag would be used when different packages provide equivalent services. For example, any package that allows a user to read mail might provide the `mail-reader` virtual package. Another package that depends on a mail reader of some sort, could require the `mail-reader` virtual package. It would then install without dependency problems, if any one of several mail programs were installed. Here's what a `provides` tag might look like:

```
Provides:  mail-reader
```

The `requires` Tag

The `requires` tag is used to alert RPM to the fact that the package needs to have certain capabilities available in order to operate properly. These capabilities refer to the name of another package, or to a virtual package provided by one or more packages that use the `provides` tag. When the `requires` tag references a package name, version comparisons may also be included by following the package name with `<`, `>`, `=`, `>=`, or `<=`, and a version specification. To get even more specific, a package's release may be included as well. Here's a `requires` tag in action, with a specific version requirement:

```
Requires:  playmidi = 2.3
```

If the `Requires` tag needs to perform a comparison against a serial number defined with the `serial` tag (described below), then the proper format would be:

```
Requires:  playmidi =S 4
```

The `conflicts` Tag

The `conflicts` tag is the logical complement to the `requires` tag. The `requires` tag is used to specify what packages *must* be present in order for the current package to operate properly. The `conflicts` tag is used to specify what packages *cannot* be installed if the current package is to operate properly.

The `conflicts` tag has the same format as the `requires` tag — namely, the tag is followed by a real or virtual package name. Like `requires`, the `conflicts` tag also accepts version and release specifications:

```
Conflicts:  playmidi = 2.3-1
```

If the `conflicts` tag needs to perform a comparision against a serial number defined with the `serial` tag (described below), then the proper format would be:

```
Conflicts:  playmidi =S 4
```

The `serial` Tag

The `serial` tag is another part of RPM's dependency and upgrade processing. The need for it is somewhat obscure, but goes something like this:

1. The package being built (call it package A) uses a version numbering scheme sufficiently obscure so that RPM cannot determine if one version is older or newer than another version.
2. Another package (package B) requires that package A be installed. More specifically, it requires RPM to compare package A's version against a specified minimum (or maximum) version.

Since RPM is unable to compare package A's version against the version specified by package B, there is no way to determine if package B's dependency requirements can be met. What to do?

The `serial` tag provides a way to get around this tricky problem. By specifying a simple integer serial number for each version, you are, in essence, directing how RPM interprets the relative age of the package. The key point to keep in mind is that in order for this to work, a unique serial number must be defined for each version of the software being packaged. In addition, the serial number must increment along with the version. Finally, the package that requires the serialized software needs to specify *its* version requirements in terms of the serial number.

Does it sound like a lot of trouble? You're right! If you find yourself in the position of needing to use this tag, take a deep breath and seriously consider changing the way you assign version numbers. If you're packaging someone else's software, perhaps you can convince them to make the change. Chances are, if RPM can't figure out the version number, most people can't, either! An example `serial` tag would look something like this:

```
Serial: 4
```

Note that RPM considers a package with a serial number as newer than a package without a serial number.

The `autoreqprov` Tag

The `autoreqprov` tag is used to control the automatic dependency processing performed when the package is being built. Normally, as each package is built, the following steps are performed:

- All executable programs being packaged are analyzed to determine their shared library requirements. These requirements are automatically added to the package's requirements.
- The soname of each shared library being packaged is automatically added to the package's list of "provides" information.

By doing this, RPM reduces the need for package builders to manually add dependency information to their packages. However, there are times when RPM's automatic dependency processing may not be desirable. In those cases the `autoreqprov` tag can be used to disable automatic dependency processing.

To disable automatic dependency processing, add the following line:

```
AutoReqProv: no
```

(The number zero may be used instead of `no`) Although RPM defaults to performing automatic dependency processing, the effect of the `autoreqprov` tag can be reversed by changing `no` to `yes`. (The number one may be used instead of `yes`)

13.2.4 Architecture- and Operating System-Specific Tags

As RPM gains in popularity, more people are putting it to work on different types of computer systems. While this would not normally be a problem, things start to get a little tricky when one of the following two situations becomes commonplace:

1. A particular operating system is ported to several different hardware platforms, or architectures.
2. A particular architecture runs several different operating systems.

The real bind hits when RPM is used to package software for several of these different system environments. Without methods of controlling the build process based on architecture and operating system, package builders that develop software for more than one architecture or operating system will have a hard time indeed. The only alternative would be to maintain parallel RPM build environments and accept all the coordination headaches that would entail.

Fortunately, RPM makes it all easier than that. With the following tags, it's possible to support package building under multiple environments, all from a single set of sources, patches, and a single spec file. For a more complete discussion of multi-architecture package building, please see chapter 19.

The `excludearch` Tag

The `excludearch` tag directs RPM to ensure that the package does *not* attempt to build on the excluded architecture(s). The reasons for preventing a package from building on a certain architecture might include:

- The software has not yet been ported to the excluded architecture.
- The software would serve no purpose on the excluded architecture.

An example of the first case might be that the software was designed based on the assumption that an integer is a 32-bit quantity. Obviously, this assumption is not valid on a 64-bit processor.

In the second case, software that depended on or manipulated low-level features of a given architecture, should be excluded from building on a different architecture. Assembly language programs would fall into this category.

One or more architectures may be specified after the `excludearch` tag, separated by either spaces or commas. Here is an example:

```
ExcludeArch: sparc alpha
```

In this example, RPM would not attempt to build the package on either the Sun SPARC or Digital Alpha/AXP architectures. The package would build on any other architectures, however. If a build is attempted on an excluded architecture, the following message will be displayed, and the build will fail:

```
# rpm -ba cdplayer-1.0.spec
Arch mismatch!
cdplayer-1.0.spec doesn't build on this architecture
```

```
#
```

Note that if your goal is to ensure that a package will only build on *one* architecture, then you should use the `exclusivearch` tag.

The `exclusivearch` Tag

The `exclusivearch` tag is used to direct RPM to ensure the package is *only* built on the specified architecture(s). The reasons for this are similar to the those mentioned in the section on the `excludearch` tag above. However, the `exclusivearch` tag is useful when the package builder needs to ensure that *only* the specified architectures will build the package. This tag ensures that no future architectures will mistakenly attempt to build the package. This would not be the case if the `excludearch` tag were used to specify every architecture known at the time the package is built.

The syntax of the `exclusivearch` tag is identical to that of `excludearch`:

```
ExclusiveArch:  sparc alpha
```

In this example, the package will only build on a Sun SPARC or Digital Alpha/AXP system.

Note that if your goal is to ensure that a package will *not* build on specific architectures, then you should use the `excludearch` tag.

The `excludeos` Tag

The `excludeos` tag is used to direct RPM to ensure that the package does *not* attempt to build on the excluded operating system(s). This is usually necessary when a package is to be built on more than one operating system, but it is necessary to keep a particular operating system from attempting a build.

Note that if your goal is to ensure that a package will only build on *one* operating system, then you should use the `exclusiveos` tag. Here's a sample `excludeos` tag:

```
ExcludeOS: linux irix
```

The `exclusiveos` Tag

The `exclusiveos` tag has the same syntax as `excludeos`, but it has the opposite logic. The `exclusiveos` tag is used to denote which operating system(s) should *only* be be permitted to build the package. Here's `exclusiveos` in action:

```
ExclusiveOS: linux
```

Note that if your goal is to ensure that a package will *not* build on a specific operating system, then you should use the `excludeos` tag.

13.2.5 Directory-related Tags

A number of tags are used to specify directories and paths that are used in various phases of RPM's build and install processes. There's not much more to say collectively about these tags, so let's dive right in and look them over.

The `prefix` Tag

The `prefix` tag is used when a relocatable package is to be built. A relocatable package can be installed normally or can be installed in a user-specified directory, by using RPM's `--prefix` install-time option. The data specified after the `prefix` tag should be the part of the package's path that may be changed during installation. For example, if the following `prefix` line was included in a spec file:

```
Prefix: /opt
```

and the following file was specified in the spec file's `%files` list:

```
/opt/blather/foonly
```

then the file `foonly` would be installed in `/opt/blather` if the package was installed normally. It would be installed in `/usr/local/blather` if the package was installed with the `--prefix /usr/local` option.

For more information about creating relocatable packages, see chapter 15.

The `buildroot` Tag

The `buildroot` tag is used to define an alternate build root. The name is a bit misleading, as the build root is actually used when the software is *installed* during the build process. In order for a build root to be defined and actually used, a number of issues must be taken into account. These issues are covered in chapter 16. This is what a `buildroot` tag would look like:

```
BuildRoot: /tmp/cdplayer
```

The `buildroot` tag can be overridden at build-time by using the `--buildroot` command-line option.

13.2.6 Source and Patch Tags

In order to build and package software, RPM needs to know where to find the original sources. But it's not quite that simple. There might be more than one set of sources that need to be part of a particular build. In some cases, it might be necessary to prevent some sources from being packaged.

And then there is the matter of patches. It's likely that changes will need to be made to the sources, so it's necessary to specify a patch file. But the same issues that apply to source specifications are also applicable to patches. There might be more than one set of patches required.

The tags that follow are crucial to RPM, so it pays to have a firm grasp of how they are used.

The `source` Tag

The `source` tag is central to nearly every spec file. Although it has only one piece of data associated with it, it actually performs two functions:

1. It shows where the software's developer has made the original sources available.
2. It gives RPM the name of the original source file.

While there is no hard and fast rule, for the first function, it's generally considered best to put this information in the form of a Uniform Resource Locator (URL). The URL should point directly to the source file itself. This is due to the `source` tag's second function.

As mentioned above, the `source` tag also needs to direct RPM to the source file on the build system. How does it do this? There's only one requirement, and it is ironclad: The source filename must be at the end of the line as the final element in a path. Here's an example:

```
Source:  ftp://ftp.gnomovision.com/pub/cdplayer-1.0.tgz
```

Given this `source` line, RPM will search its `SOURCES` directory for `cdplayer-1.0.tgz`. Everything prior to the filename is ignored by RPM. It's there strictly for any interested humans.

A spec file may contain more than one `source` tag. This is necessary for those cases where the software being packaged is contained in more than one source file. However, the `source` tags must be uniquely identified. This is done by appending a number to the end of the tag itself. In fact, RPM does this internally for the first `source` tag in a spec file, in essence turning it into `source0`. Therefore, if a package contains two source files, they may either be specified as:

```
Source: blather-4.5.tar.gz
Source1: bother-1.2.tar.gz
```

or as:

```
Source0: blather-4.5.tar.gz
Source1: bother-1.2.tar.gz
```

Either approach may be used. The choice is yours.

The `nosource` Tag

The `nosource` tag is used to direct RPM to omit one or more source files from the source package. Why would someone want to go to the trouble of specifying a source file, only to exclude it? The reasons for this can be varied, but let's look at one example: The software known as Pretty Good Privacy, or PGP.

PGP contains encryption routines of such high quality that the United States government restricts their export.[1] While it would be nice to create a PGP package file, the resulting package could not legally be transferred between the U.S. and other countries, or vice-versa.

However, what if all files other than the original source, were packaged using RPM? Well, a binary package made without PGP would be of little use, but what about the source package? It would contain the spec file, maybe some patches, and perhaps even an icon file. Since the controversial PGP software was not a part of the source package, this sanitized source package could be downloaded legally in any country. The person that downloaded a copy could then go about legally obtaining the PGP sources themselves, place them in RPM's `SOURCES` directory, and create a binary package. They wouldn't even need to change the `nosource` tag. One `rpm -ba` command later, and the user would have a perfectly usable PGP binary package file.

Since there may be more than one `source` tag in a spec file, the format of the `nosource` tag is as follows:

```
nosource: <src-num>, <src-num>...<src-num>
```

The `<src-num>` represents the number following the `source` tag. If there is more than one number in the list, they may be separated by either commas or spaces. For example, consider a package containing the following `source` tags:

```
source: blather-4.5.tar.gz
Source1: bother-1.2.tar.gz
source2: blather-lib-4.5.tar.gz
source3: bother-lib-1.2.tar.gz
```

If the source files for `blather` and `blather-lib` were not to be included in the package, the following `nosource` line could be added:

```
NoSource:  0, 3
```

What about that `0`? Keep in mind that the first unnumbered `source` tag in a spec file is automatically numbered 0 by RPM.

The `patch` Tag

The `patch` tag is used to identify which patches are associated with the software being packaged. The patch files are kept in RPM's `SOURCES` directory, so only the name of the patch file should be specified. Here is an example:

```
Patch: cdp-0.33-fsstnd.patch
```

[1] There is also an "international" version that may be used in non-US countries. See Appendix G on page 427.

There are no hard and fast requirements for naming the patch files, but traditionally the filename starts with the software name and version, separated by dashes. The next part of the patch file name usually includes one or more words indicating the reason for the patch. In our example above, the patch file contains changes necessary to bring the software into compliance with the Linux File System Standard, hence the `fsstnd` magic incantation.

RPM processes `patch` tags the same way it does `source` tags. Therefore, it's acceptable to use a Uniform Resource Locator (URL) on a `patch` line, too.

A spec file may contain more than one `patch` tag. This is necessary for those cases where the software being packaged requires more than one patch. However, the `patch` tags must be uniquely identified. This is done by appending a number to the end of the tag itself. In fact, RPM does this internally for the first `patch` tag in a spec file, in essence turning it into `patch0`. Therefore, if a package contains three patches, the following two methods of specifying them are equivalent:

```
Patch: blather-4.5-bugfix.patch
Patch1: blather-4.5-config.patch
Patch2: blather-4.5-somethingelse.patch
```

This is the same as:

```
Patch0: blather-4.5-bugfix.patch
Patch1: blather-4.5-config.patch
Patch2: blather-4.5-somethingelse.patch
```

Either approach may be used, but the second method looks nicer.

The `nopatch` Tag

The `nopatch` tag is similar to the `nosource` tag discussed earlier. Just like the `nosource` tag, the `nopatch` tag is used to direct RPM to omit something from the source package. In the case of `nosource`, that "something" was one or more sources. For the `nopatch` tag, the "something" is one or more patches.

Since each `patch` tag in a spec file must be numbered, the `nopatch` tag uses those numbers to specify which patches are not to be included in the package. The `nopatch` tag is used in this manner:

```
NoPatch:  2 3
```

In this example, the source files specified on the `source2` and `source3` lines are not to be included in the build.

This concludes our study of RPM's tags. In the next section, we'll look at the various scripts that RPM uses to build, as well as to install, and erase, packages.

13.3 Scripts: RPM's Workhorse

The scripts that RPM uses to control the build process are among the most varied and interesting parts of the spec file. Many spec files also contain scripts that perform a variety of tasks whenever the package is installed or erased.

The start of each script is denoted by a keyword. For example, the `%build` keyword marks the start of the script RPM will execute when building the software to be packaged. It should be noted that, in the strictest sense of the word, these parts of the spec file are not scripts. For example, they do not start with the traditional invocation of a shell. However, the contents of each script section are copied into a file and executed by RPM as a full-fledged script. This is part of the power of RPM: Anything that can be done in a script can be done by RPM.

Let's start by looking at the scripts used during the build process.

13.3.1 Build-time Scripts

The scripts that RPM uses during the building of a package follow the steps known to every software developer:

- Unpacking the sources.
- Building the software.
- Installing the software.
- Cleaning up.

Although each of the scripts perform a specific function in the build process, they share a common environment. Using RPM's `--test` option[2], we can see the common portion of each script. In the following example, we've taken the `cdplayer` package, issued an `rpm -ba --test cdplayer-1.0-1.spec`, and viewed the script files left in RPM's temporary directory. This section (with the appropriate package-specific values) is present in every script RPM executes during a build:

```
#!/bin/sh -e
# Script generated by rpm

RPM_SOURCE_DIR="/usr/src/redhat/SOURCES"
RPM_BUILD_DIR="/usr/src/redhat/BUILD"
RPM_DOC_DIR="/usr/doc"
RPM_OPT_FLAGS="-O2 -m486 -fno-strength-reduce"
RPM_ARCH="i386"
RPM_OS="Linux"
RPM_ROOT_DIR="/tmp/cdplayer"
```

[2]Described in section 12.1.11 on page 154.

```
RPM_BUILD_ROOT="/tmp/cdplayer"
RPM_PACKAGE_NAME="cdplayer"
RPM_PACKAGE_VERSION="1.0"
RPM_PACKAGE_RELEASE="1"
set -x

umask 022
```

As we can see, the script starts with the usual invocation of a shell (in this case, the Bourne shell). There are no arguments passed to the script. Next, a number of environment variables are set. Here's a brief description of each one:

- `RPM_SOURCE_DIR` — This environment variable gets its value from the rpmrc file entry `sourcedir`, which in turn can get part of its value from the `topdir` entry. It is the path RPM will prepend to the file, specified in the `source` tag line.
- `RPM_BUILD_DIR` — This variable is based on the `builddir` rpmrc file entry, which in turn can get part of its value from the `topdir` entry. This environment variable translates to the path of RPM's build directory, where most software will be unpacked and built.
- `RPM_DOC_DIR` — The value of this environment variable is based on the `defaultdocdir` `rpmrc` file entry. Files marked with the `%doc` directive can be installed in a subdirectory of `defaultdocdir`. For more information on the `%doc` directive, refer to section 13.6.1 on page 199.
- `RPM_OPT_FLAGS` — This environment variable gets its value from the `optflags` `rpmrc` file entry. It contains options that can be passed on to the build procedures of the software being packaged. Normally this means either a configuration script or the `make` command itself.
- `RPM_ARCH` — As one might infer from the example above, this environment variable contains a string describing the build system's architecture.
- `RPM_OS` — This one contains the name of the build system's operating system.
- `RPM_BUILD_ROOT` — This environment variable is used to hold the "build root", into which the newly built software will be installed. If no explicit build root has been specified (either by command line option, spec file tag line, or rpmrc file entry), this variable will be null.
- `RPM_PACKAGE_NAME` — This environment variable gets its value from the `name` tag line in the package's spec file. It contains the name of the software being packaged.
- `RPM_PACKAGE_VERSION` — The `version` tag line is the source of this variable's translation. Predictably, this environment variable contains the software's version number.
- `RPM_PACKAGE_RELEASE` — This environment variable contains the package's release number. Its value is obtained from the `release` tag line in the spec file.

All of these environment variables are set for your use, to make it easier to write scripts that will do the right thing even if the build environment changes.

The script also sets an option that causes the shell to print out each command, complete with expanded arguments. Finally, the default permissions are set. Past this point, the scripts differ. Let's look at the scripts in the order they are executed.

The `%prep` Script

The `%prep` script is the first script RPM executes during a build. Prior to the `%prep` script, RPM has performed preliminary consistency checks, such as whether the spec file's `source` tag points to files that actually exist. Just prior to passing control over to the `%prep` script's contents, RPM changes directory into RPM's build area, which, by default, is `/usr/src/redhat/BUILD`.

At that point, it is the responsibility of the `%prep` script to:

- Create the top-level build directory.
- Unpack the original sources into the build directory.
- Apply patches to the sources, if necessary.
- Perform any other actions required to get the sources in a ready-to-build state.

The first three items on this list are common to the vast majority of all software being packaged. Because of this, RPM has two macros that greatly simplify these routine functions. More information on RPM's `%setup` and `%patch` macros can be found in section 13.4 on page 186.

The last item on the list can include creating directories or anything else required to get the sources in a ready-to-build state. As a result, a `%prep` script can range from one line invoking a single `%setup` macro, to many lines of tricky shell programming.

The `%build` Script

The `%build` script picks up where the `%prep` script left off. Once the `%prep` script has gotten everything ready for the build, the `%build` script is usually somewhat anti-climactic — normally invoking `make`, maybe a configuration script, and little else.

Like `%prep` before it, the `%build` script has the same assortment of environment variables to draw on. Also, like `%prep`, `%build` changes directory into the software's top-level build directory (located in `RPM_BUILD_DIR`, or usually called *<name>-<version>*).

Unlike `%prep`, there are no macros available for use in the `%build` script. The reason is simple: Either the commands required to build the software are simple (such as a single `make` command), or they are so unique that a macro wouldn't make it easier to write the script.

The %install Script

The environment in which the %install script executes is identical to the other scripts. Like the other scripts, the %install script's working directory is set to the software's top-level directory.

As the name implies, it is this script's responsibility to do whatever is necessary to actually install the newly built software. In most cases, this means a single make install command, or a few commands to copy files and create directories.

The %clean Script

The %clean script, as the name implies, is used to clean up the software's build directory tree. RPM normally does this for you, but in certain cases (most notably in those packages that use a build root) you'll need to include a %clean script.

As usual, the %clean script has the same set of environment variables as the other scripts we've covered here. Since a %clean script is normally used when the package is built in a build root, the RPM_BUILD_ROOT environment variable is particularly useful. In many cases, a simple

```
rm -rf $RPM_BUILD_ROOT
```

will suffice.[3]

13.3.2 Install/Erase-time Scripts

The other type of scripts that are present in the spec file are those that are only used when the package is either installed or erased. There are four scripts, each one meant to be executed at different times during the life of a package:

- Before installation.
- After installation.
- Before erasure.
- After erasure.

Unlike the build-time scripts, there is little in the way of environment variables for these scripts. The only environment variable available is RPM_INSTALL_PREFIX, and that is only set if the package uses an installation prefix.

Unlike the build-time scripts, there *is* an argument defined. The sole argument to these scripts, is a number representing the number of instances of the package currently installed

[3]Keep in mind that this command in a %clean script can wreak havoc if used with a build root of, say, /. Section 12.1.13 on page 158 discusses this in more detail.

on the system, *after* the current package has been installed or erased. Sound tricky? It really isn't. Here's an example:

Assume that a package, called `blather-1.0`, is being installed. No previous versions of `blather` have been installed. Since the software is being installed, only the `%pre` and `%post` scripts are executed. The argument passed to these scripts will be 1, since the the number of `blather` packages installed is 1.[4]

Continuing our example, a new version of the `blather` package, version 1.3, is now available. Clearly it's time to upgrade. What will the scripts' values be during the upgrade? As `blather-1.3` is installing, its `%pre` and `%post` scripts will have an argument equal to 2 (1 for version 1.0 already installed, plus 1 for version 1.3 being installed). As the final part of the upgrade, it's then time to erase `blather` version 1.0. As the package is being removed, its `%preun` and `%postun` scripts are executed. Since there will be only one `blather` package (version 1.3) installed after version 1.0 is erased, the argument passed to version 1.0's scripts is 1.

To finally bring an end to this example, we've decided to erase `blather` 1.3. We just don't need it anymore. As the package is being erased, its `%preun` and `%postun` scripts will be executed. Since there will be no `blather` packages installed once the erase completes, the argument passed to the scripts is 0.

With all that said, of what possible use would this argument be? Well, it has two very interesting properties:

1. When the first version of a package is installed, its `%pre` and `%post` scripts will be passed an argument equal to 1.

2. When the last version of a package is erased, its `%preun` and `%postun` scripts will be passed an argument equal to 0.

Based on these properties, it's trivial to write an install-time script that can take certain actions in specific circumstances. Usually, the argument is used in the `%preun` or `%postun` scripts to perform a special task when the last instance of a package is being erased.

What is normally done during these scripts? The exact tasks may vary, but in general, the tasks are any that need to be performed at these points in the package's existence. One very common task is to run `ldconfig` when shared libraries are installed or removed. But that's not the only use for these scripts. It's even possible to use the scripts to perform tests to ensure the package install/erasure should proceed.

Since each of these scripts will be executing on whatever system installs the package, it's necessary to choose the script's choice of tools carefully. Unless you're sure a given program is going to be available on *all* the systems that could possibly install your package, you should not use it in these scripts.

[4] Or it will be 1, once the package is completely installed. Remember, the number is based on the number of packages installed *after* the current package's install or erase has completed.

The %pre Script

The `%pre` script executes just before the package is to be installed. It is the rare package that requires anything to be done prior to installation; none of the 350 packages that comprise Red Hat Linux 4.0 make use of it.

The %post Script

The `%post` script executes after the package has been installed. One of the most popular reasons a `%post` script is needed is to run `ldconfig` to update the list of available shared libraries after a new one has been installed. Of course, other functions can be performed in a `%post` script. For example, packages that install shells use the `%post` script to add the shell name to `/etc/shells`.

If a package uses a `%post` script to perform some function, quite often it will include a `%postun` script that performs the inverse of the `%post` script, after the package has been removed.

The %preun Script

If there's a time when your package needs to have one last look around before the user erases it, the place to do it is in the `%preun` script. Anything that a package needs to do immediately prior to RPM taking any action to erase the package, can be done here.

The %postun Script

The `%postun` script executes after the package has been removed. It is the last chance for a package to clean up after itself. Quite often, `%postun` scripts are used to run `ldconfig` to remove newly erased shared libraries from `ld.so.cache`.

13.3.3 Verification-Time Script — The %verifyscript Script

The `%verifyscript` executes whenever the installed package is verified by RPM's verification command. The contents of this script is entirely up to the package builder, but in general the script should do whatever is necessary to verify the package's proper installation. Since RPM automatically verifies the existence of a package's files, along with other file attributes, the `%verifyscript` should concentrate on different aspects of the package's installation. For example, the script may ensure that certain configuration files contain the proper information for the package being verified:

```
for n in ash bsh; do
    echo -n "Looking for $n in /etc/shells... "
    if ! grep "^/bin/${n}\$" /etc/shells > /dev/null; then
        echo "missing"
        echo "${n} missing from /etc/shells" >&2
```

```
    else
        echo "found"
    fi
done
```

In this script, the config file `/etc/shells`, is checked to ensure that it has entries for the shells provided by this package.

It is worth noting that the script sends informational and error messages to stdout, and error messages only to stderr. Normally RPM will only display error output from a verification script; the output sent to stdout is only displayed when the verification is run in verbose mode.

13.4 Macros: Helpful Shorthand for Package Builders

RPM does not support macros in the sense of ad-hoc sequences of commands being defined as a macro and executed by simply referring to the macro name.

However, there are two parts of RPM's build process that are fairly constant from one package to another, and they are the unpacking and patching of sources. Because of this, RPM makes two macros available to simplify these tasks:

1. The `%setup` macro, which is used to unpack the original sources.
2. The `%patch` macro, which is used to apply patches to the original sources.

These macros are used exclusively in the `%prep` script; it wouldn't make sense to use them anywhere else. The use of these macros is not mandatory — It is certainly possible to write a `%prep` script without them. But in the vast majority of cases they make life easier for the package builder.

13.4.1 The `%setup` Macro

As we mentioned above, the `%setup` macro is used to unpack the original sources, in preparation for the build. In its simplest form, the macro is used with no options and gets the name of the source archive from the `source` tag specified earlier in the spec file. Let's look at an example. The `cdplayer` package has the following `source` tag:

```
Source:  ftp://ftp.gnomovision.com/pub/cdplayer/cdplayer-1.0.tgz
```

and the following `%prep` script:

```
%prep
%setup
```

In this simple case, the `%setup` macro expands into the following commands:

```
cd /usr/src/redhat/BUILD
rm -rf cdplayer-1.0
gzip -dc /usr/src/redhat/SOURCES/cdplayer-1.0.tgz | tar -xvvf -
if [ $? -ne 0 ]; then
  exit $?
fi
cd cdplayer-1.0
cd /usr/src/redhat/BUILD/cdplayer-1.0
chown -R root.root .
chmod -R a+rX,g-w,o-w .
```

As we can see, the `%setup` macro starts by changing directory into RPM's build area and removing any `cdplayer` build trees from previous builds. It then uses `gzip` to uncompress the original source (whose name was taken from the `source` tag), and pipes the result to `tar` for unpacking. The return status of the unpacking is tested. If sucessful, the macro continues.

At this point, the original sources have been unpacked. The `%setup` macro continues by changing directory into `cdplayer`'s top-level directory. The two `cd` commands are an artifact of `%setup`'s macro expansion. Finally, `%setup` makes sure every file in the build tree is owned by root and has appropriate permissions set.

But that's just the simplest way that `%setup` can be used. There are a number of other options that can be added to accomodate different situations. Let's look at them.

`-n` *<name>* – Set Name of Build Directory

In our example above, the `%setup` macro simply uncompressed and unpacked the sources. In this case, the `tar` file containing the original sources was created such that the top-level directory was included in the tar file. The name of the top-level directory was also identical to that of the `tar` file, which was in *<name>-<version>* format.

However, this is not always the case. Quite often, the original sources unpack into a directory whose name is different than the original `tar` file. Since RPM assumes the directory will be called *<name>-<version>*, when the directory is called something else, it's necessary to use `%setup`'s `-n` option. Here's an example:

Assume, for a moment, that the `cdplayer` sources, when unpacked, create a top-level directory named `cd-player`. In this case, our `%setup` line would look like this:

`%setup -n cd-player`

and the resulting commands would look like this:

```
cd /usr/src/redhat/BUILD
rm -rf cd-player
gzip -dc /usr/src/redhat/SOURCES/cdplayer-1.0.tgz | tar -xvvf -
if [ $? -ne 0 ]; then
  exit $?
fi
```

```
cd cd-player
cd /usr/src/redhat/BUILD/cd-player
chown -R root.root .
chmod -R a+rX,g-w,o-w .
```

The results are identical to using `%setup` with no options, except for the fact that `%setup` now does a recursive delete on the directory `cd-player` (instead of `cdplayer-1.0`), and changes directory into `cd-player` (instead of `cdplayer-1.0`).

Note that all subsequent build-time scripts will change directory into the directory specified by the `-n` option. This makes `-n` unsuitable as a means of unpacking sources in directories other than the top-level build directory. In the upcoming example on page 191, we'll show a way around this restriction.

A quick word of warning: If the name specified with the `-n` option doesn't match the name of the directory created when the sources are unpacked, the build will stop pretty quickly, so it pays to be careful when using this option.

`-c` – Create Directory (and change to it) Before Unpacking

How many times have you grabbed a `tar` file and unpacked it, only to find that it splattered files all over your current directory? Sometimes source archives are created without a top-level directory.

As you can see from the examples so far, `%setup` expects the archive to create its own top-level directory. If this isn't the case, you'll need to use the `-c` option.

This option simply creates the directory and changes directory into it before unpacking the sources. Here's what it looks like:

```
cd /usr/src/redhat/BUILD
rm -rf cdplayer-1.0
mkdir -p cdplayer-1.0
cd cdplayer-1.0
gzip -dc /usr/src/redhat/SOURCES/cdplayer-1.0.tgz | tar -xvvf -
if [ $? -ne 0 ]; then
  exit $?
fi
cd /usr/src/redhat/BUILD/cdplayer-1.0
chown -R root.root .
chmod -R a+rX,g-w,o-w .
```

The only changes from using `%setup` with no options, are the `mkdir` and `cd` commands, prior to the commands that unpack the sources. Note that you can use the `-n` option along with `-c`, so something like `%setup -c -n blather` works as expected.

-D – Do Not Delete Directory Before Unpacking Sources

The `-D` option keeps the `%setup` macro from deleting the software's top-level directory. This option is handy when the sources being unpacked are to be added to an already-existing directory tree. This would be the case when more than one `%setup` macro is used. Here's what `%setup` does when the `-D` option is employed:

```
cd /usr/src/redhat/BUILD
gzip -dc /usr/src/redhat/SOURCES/cdplayer-1.0.tgz | tar -xvvf -
if [ $? -ne 0 ]; then
  exit $?
fi
cd cdplayer-1.0
cd /usr/src/redhat/BUILD/cdplayer-1.0
chown -R root.root .
chmod -R a+rX,g-w,o-w .
```

As advertised, the `rm` prior to the `tar` command is gone.

-T – Do Not Perform Default Archive Unpacking

The `-T` option disables `%setup`'s normal unpacking of the archive file specified on the `source0` line. Here's what the resulting commands look like:

```
cd /usr/src/redhat/BUILD
rm -rf cdplayer-1.0
cd cdplayer-1.0
cd /usr/src/redhat/BUILD/cdplayer-1.0
chown -R root.root .
chmod -R a+rX,g-w,o-w .
```

Doesn't make much sense, does it? There's a method to this madness. We'll see the `-T` in action in the next section.

-b *<n>* — Unpack The *n*th Sources Before Changing Directory

The `-b` option is used in conjunction with the `source` tag. Specifically, it is used to identify which of the numbered `source` tags in the spec file are to be unpacked.

The `-b` option requires a numeric argument matching an existing `source` tag. If a numeric argument is not provided, the build will fail:

```
# rpm -ba cdplayer-1.0.spec
Package: cdplayer
Need arg to %setup -b
Build failed.
```

```
#
```

Remembering that the first `source` tag is implicitly numbered 0, let's see what happens when the `%setup` line is changed to `%setup -b 0`:

```
cd /usr/src/redhat/BUILD
rm -rf cdplayer-1.0
gzip -dc /usr/src/redhat/SOURCES/cdplayer-1.0.tgz | tar -xvvf -
if [ $? -ne 0 ]; then
  exit $?
fi
gzip -dc /usr/src/redhat/SOURCES/cdplayer-1.0.tgz | tar -xvvf -
if [ $? -ne 0 ]; then
  exit $?
fi
cd cdplayer-1.0
cd /usr/src/redhat/BUILD/cdplayer-1.0
chown -R root.root .
chmod -R a+rX,g-w,o-w .
```

That's strange. The sources were unpacked twice. It doesn't make sense, until you realize that this is why there is a `-T` option. Since `-T` disables the default source file unpacking, and `-b` selects a particular source file to be unpacked, the two are meant to go together, like this:

```
%setup -T -b 0
```

Looking at the resulting commands, we find:

```
cd /usr/src/redhat/BUILD
rm -rf cdplayer-1.0
gzip -dc /usr/src/redhat/SOURCES/cdplayer-1.0.tgz | tar -xvvf -
if [ $? -ne 0 ]; then
  exit $?
fi
cd cdplayer-1.0
cd /usr/src/redhat/BUILD/cdplayer-1.0
chown -R root.root .
chmod -R a+rX,g-w,o-w .
```

That's more like it! Let's go on to the next option.

`-a` *<n>* — Unpack The *n*th Sources After Changing Directory

The `-a` option works similarly to the `-b` option, except that the sources are unpacked *after* changing directory into the top-level build directory. Like the `-b` option, `-a` requires `-T` in order to prevent two sets of unpacking commands. Here are the commands that a `%setup -T -a 0` line would produce:

```
cd /usr/src/redhat/BUILD
rm -rf cdplayer-1.0
cd cdplayer-1.0
gzip -dc /usr/src/redhat/SOURCES/cdplayer-1.0.tgz | tar -xvvf -
if [ $? -ne 0 ]; then
  exit $?
fi
cd /usr/src/redhat/BUILD/cdplayer-1.0
chown -R root.root .
chmod -R a+rX,g-w,o-w .
```

Note that there is no `mkdir` command to create the top-level directory prior to issuing a `cd` into it. In our example, adding the `-c` option will make things right:

```
cd /usr/src/redhat/BUILD
rm -rf cdplayer-1.0
mkdir -p cdplayer-1.0
cd cdplayer-1.0
gzip -dc /usr/src/redhat/SOURCES/cdplayer-1.0.tgz | tar -xvvf -
if [ $? -ne 0 ]; then
  exit $?
fi
cd /usr/src/redhat/BUILD/cdplayer-1.0
chown -R root.root .
chmod -R a+rX,g-w,o-w .
```

The result is the proper sequence of commands for unpacking a `tar` file with no top-level directory.

Using `%setup` in a Multi-source Spec File

If all these interrelated options seem like overkill for unpacking a single source file, you're right. The real reason for the various options is to make it easier to combine several separate source archives into a single, build-able entity. Let's see how they work in that type of environment.

For the purposes of this example, our spec file will have the following three `source` tags:[5]

```
source: source-zero.tar.gz
source1: source-one.tar.gz
source2: source-two.tar.gz
```

[5] Yes, the `source` tags should include a URL pointing to the sources.

To unpack the first source is not hard; all that's required is to use `%setup` with no options:

```
%setup
```

This produces the following set of commands:

```
cd /usr/src/redhat/BUILD
rm -rf cdplayer-1.0
gzip -dc /usr/src/redhat/SOURCES/source-zero.tar.gz | tar -xvvf -
if [ $? -ne 0 ]; then
  exit $?
fi
cd cdplayer-1.0
cd /usr/src/redhat/BUILD/cdplayer-1.0
chown -R root.root .
chmod -R a+rX,g-w,o-w .
```

If `source-zero.tar.gz` didn't include a top-level directory, we could have made one by adding the `-c` option:

```
%setup -c
```

which would result in:

```
cd /usr/src/redhat/BUILD
rm -rf cdplayer-1.0
mkdir -p cdplayer-1.0
cd cdplayer-1.0
gzip -dc /usr/src/redhat/SOURCES/source-zero.tar.gz | tar -xvvf -
if [ $? -ne 0 ]; then
  exit $?
fi
cd /usr/src/redhat/BUILD/cdplayer-1.0
chown -R root.root .
chmod -R a+rX,g-w,o-w .
```

Of course, if the top-level directory did not match the package name, the `-n` option could have been added:

```
%setup -n blather
```

which results in:

```
cd /usr/src/redhat/BUILD
rm -rf blather
gzip -dc /usr/src/redhat/SOURCES/source-zero.tar.gz | tar -xvvf -
if [ $? -ne 0 ]; then
  exit $?
```

```
fi
cd blather
cd /usr/src/redhat/BUILD/blather
chown -R root.root .
chmod -R a+rX,g-w,o-w .
```

or

```
%setup -c -n blather
```

This results in:

```
cd /usr/src/redhat/BUILD
rm -rf blather
mkdir -p blather
cd blather
gzip -dc /usr/src/redhat/SOURCES/source-zero.tar.gz | tar -xvvf -
if [ $? -ne 0 ]; then
  exit $?
fi
cd /usr/src/redhat/BUILD/blather
chown -R root.root .
chmod -R a+rX,g-w,o-w .
```

Now let's add the second source file. Things get a bit more interesting here. First, we need to identify which `source` tag (and therefore, which source file) we're talking about. So we need to use either the `-a` or `-b` option, depending on the characteristics of the source archive. For this example, let's say that `-a` is the option we want. Adding that option, plus a "1" to point to the source file specified in the `source1` tag, we have:

```
%setup -a 1
```

Since we've already seen that using the `-a` or `-b` option results in duplicate unpacking, we need to disable the default unpacking by adding the `-T` option:

```
%setup -T -a 1
```

Next, we need to make sure that the top-level directory isn't deleted. Otherwise, the first source file we just unpacked would be gone. That means we need to include the `-D` option to prevent that from happening. Adding this final option, and including the now complete macro in our `%prep` script, we now have:

```
%setup
%setup -T -D -a 1
```

This will result in the following commands:

```
cd /usr/src/redhat/BUILD
```

```
rm -rf cdplayer-1.0
gzip -dc /usr/src/redhat/SOURCES/source-zero.tar.gz | tar -xvvf -
if [ $? -ne 0 ]; then
  exit $?
fi
cd cdplayer-1.0
cd /usr/src/redhat/BUILD/cdplayer-1.0
chown -R root.root .
chmod -R a+rX,g-w,o-w .
cd /usr/src/redhat/BUILD
cd cdplayer-1.0
gzip -dc /usr/src/redhat/SOURCES/source-one.tar.gz | tar -xvvf -
if [ $? -ne 0 ]; then
  exit $?
fi
cd /usr/src/redhat/BUILD/cdplayer-1.0
chown -R root.root .
chmod -R a+rX,g-w,o-w .
```

So far, so good. Let's include the last source file, but with this one, we'll say that it needs to be unpacked in a subdirectory of `cdplayer-1.0` called `database`. Can we use `%setup` in this case?

We could, if `source-two.tgz` created the `database` subdirectory. If not, then it'll be necessary to do it by hand. For the purposes of our example, let's say that `source-two.tgz` wasn't created to include the `database` subdirectory, so we'll have to do it ourselves. Here's our `%prep` script now:

```
%setup
%setup -T -D -a 1
mkdir database
cd database
gzip -dc /usr/src/redhat/SOURCES/source-two.tar.gz | tar -xvvf -
```

Here's the resulting script:

```
cd /usr/src/redhat/BUILD
rm -rf cdplayer-1.0
gzip -dc /usr/src/redhat/SOURCES/source-zero.tar.gz | tar -xvvf -
if [ $? -ne 0 ]; then
  exit $?
fi
cd cdplayer-1.0
cd /usr/src/redhat/BUILD/cdplayer-1.0
chown -R root.root .
chmod -R a+rX,g-w,o-w .
cd /usr/src/redhat/BUILD
```

```
cd cdplayer-1.0
gzip -dc /usr/src/redhat/SOURCES/source-one.tar.gz | tar -xvvf -
if [ $? -ne 0 ]; then
  exit $?
fi
mkdir database
cd database
gzip -dc /usr/src/redhat/SOURCES/source-two.tar.gz | tar -xvvf -
```

The three commands we added to unpack the last set of sources were added to the end of the `%prep` script.

The bottom line to using the `%setup` macro is that you can probably get it to do what you want, but don't be afraid to tinker. And even if `%setup` can't be used, it's easy enough to add the necessary commands to do the work manually. Above all, make sure you use the `--test` option when testing your `%setup` macros, so you can see what commands they're translating to.

Next, let's look at RPM's second macro, `%patch`.

13.4.2 The `%patch` Macro

The `%patch` macro, as its name implies, is used to apply patches to the unpacked sources. In the following examples, our spec file has the following `patch` tag lines:

```
patch0: patch-zero
patch1: patch-one
patch2: patch-two
```

At its simplest, the `%patch` macro can be invoked without any options:

```
%patch
```

Here are the resulting commands:

```
echo "Patch #0:"
patch -p0  -s < /usr/src/redhat/SOURCES/patch-zero
```

The `%patch` macro nicely displays a message showing that a patch is being applied, then invokes the `patch` command to actually do the dirty work. There are two options to the `patch` command:

1. The `-p` option, which directs `patch` to remove the specified number of slashes (and any intervening directories) from the front of any filenames specified in the patch file. In this case, nothing will be removed.
2. The `-s` option, which directs `patch` to apply the patch without displaying any informational messages. Only errors from `patch` will be displayed.

How did the `%patch` macro know which patch to apply? Keep in mind that, like the `source` tag lines, every `patch` tag is numbered, starting at zero. The `%patch` macro, by default, applies the patch file named on the `patch` (or `patch0`) tag line.

Specifying Which `patch` Tag to Use

The `%patch` macro actually has two different ways to specify the `patch` tag line it is to use. The first method is to simply append the number of the desired `patch` tag to the end of the `%patch` macro itself. For example, in order to apply the patch specified on the `patch2` tag line, the following `%patch` macro could be used:

```
%patch2
```

The other approach is to use the `-P` option. This option is followed by the number of the `patch` tag line desired. Therefore, this line is identical in function to the previous one:

```
%patch -P 2
```

Note that the `-P` option will *not* apply the file specified on the `patch0` line, by default. Therefore, if you choose to use the `-P` option to specify patch numbers, you'll need to use the following format when applying patch zero:

```
%patch -P 0
```

`-p <#>` – Strip <#> leading slashes and directories from patch filenames

The `-p` (Note the *lowercase* "p"!) option is sent directly to the `patch` command. It is followed by a number, which specifies the number of leading slashes (and the directories in between) to strip from any filenames present in the patch file. For more information on this option, please consult the `patch` man page.

`-b` *<name>* – Set the backup file extension to *<name>*

When the `patch` command is used to apply a patch, unmodified copies of the files patched are renamed to end with the extension `.orig`. The `-b` option is used to change the extension used by `patch`. This is normally done when multiple patches are to be applied to a given file. By doing this, copies of the file as it existed prior to each patch, are readily available.

`-E` — Remove Empty Output Files

The `-E` option is passed directly to the `patch` program. When `patch` is run with the `-E` option, any output files that are empty after the patches have been applied, are removed.

Now let's take `%patch` on a test-drive, and put it through its paces.

An example of the `%patch` Macro in Action

Using the example `patch` tag lines we've used throughout this section, let's put together an example and look at the resulting commands. In our example, the first patch to be applied needs to have the root directory stripped. Its `%patch` macro will look like this:

```
%patch -p1
```

The next patch is to be applied to files in the software's `lib` subdirectory, so we'll need to add a `cd` command to get us there. We'll also need to strip an additional directory:

```
cd lib
%patch -P 1 -p2
```

Finally, the last patch is to be applied from the software's top-level directory, so we need to `cd` back up a level. In addition, this patch modifies some files that were also patched the first time, so we'll need to change the backup file extension:

```
cd ..
%patch -P 2 -p1 -b .last-patch
```

Here's what the `%prep` script (minus any `%setup` macros) looks like:

```
%patch -p1
cd lib
%patch -P 1 -p2
cd ..
%patch -P 2 -p1 -b .last-patch
```

And here's what the macros expand to:

```
echo "Patch #0:"
patch -p1  -s < /usr/src/redhat/SOURCES/patch-zero
cd lib
echo "Patch #1:"
patch -p2  -s < /usr/src/redhat/SOURCES/patch-one
cd ..
echo "Patch #2:"
patch -p1 -b .last-patch -s < /usr/src/redhat/SOURCES/patch-two
```

No surprises here. Note that the `%setup` macro leaves the current working directory set to the software's top-level directory, so our `cd` commands with their relative paths will do the right thing. Of course, we have environment variables available that could be used here, too.

Compressed Patch Files If a patch file is compressed with `gzip`, RPM will automatically decompress it before applying the patch. Here's a compressed patch file as specified in the spec file:

```
Patch: bother-3.5-hack.patch.gz
```

This is part of the script RPM will execute when the `%prep` section is executed:

```
echo Executing: %prep
...
echo "Patch #0:"
gzip -dc /usr/src/redhat/SOURCES/bother-3.5-hack.patch.gz | patch -p1  -s
...
```

First, the patch file is decompressed using `gzip`. The output from `gzip` is then piped into `patch`.

That's about it for RPM's macros. Next, let's take a look at the `%files` list.

13.5 The %files List

The `%files` list indicates to RPM which files on the build system are to be packaged. The list consists of one file per line. The file may have one or more directives preceeding it. These directives give RPM additional information about the file and are discussed more fully below.

Normally, each file includes its full path. The path performs two functions. First, it specifies the file's location on the build system. Second, it denotes where the file should be placed when the package is to be installed.[6]

For packages that create directories containing hundreds of files, it can be quite cumbersome creating a list that contains every file. To make this situation a bit easier, if the `%files` list contains a path to a directory, RPM will automatically package every file in that directory, as well as every file in each subdirectory. Shell-style globbing can also be used in the `%files` list.

13.6 Directives For the %files list

The `%files` list may contain a number of different directives. They are used to:

- Identify documentation and configuration files.
- Ensure that a file has the correct permissions and ownership set.
- Control which aspects of a file are to be checked during package verification.
- Eliminate some of the tedium in creating the `%files` list.

[6]This is not entirely the case when a relocatable package is being built. For more information on relocatable packages, see chapter 15 on page 219.

In the `%files` list, one or more directives may be placed on a line, separated by spaces, before one or more filenames. Therefore, if `%foo` and `%bar` are two `%files` list directives, they may be applied to a file `baz` in the following manner:

```
%foo %bar baz
```

Now it's time to take a look at the directives that inhabit the `%files` list.

13.6.1 File-related Directives

RPM processes files differently according to their type. However, RPM does not have a method of automatically determining file types. Therefore, it is up to the package builder to appropriately mark files in the `%files` list. This is done using one of the directives below.

Keep in mind that not every file will need to be marked. As you read the following sections, you'll see that directives are only used in special circumstances. In most packages, the majority of files in the `%files` list will not need to be marked.

The `%doc` Directive

The `%doc` directive flags the filename(s) that follow, as being documentation. RPM keeps track of documentation files in its database, so that a user can easily find information about an installed package. In addition, RPM can create a package-specific documentation directory during installation and copy documentation into it. Whether or not this additional step is taken, is dependent on how a file is specified. Here is an example:

```
%doc README
%doc /usr/local/foonly/README
```

The file `README` exists in the software's top-level directory during the build, and is included in the package file. When the package is installed, RPM creates a directory in the documentation directory named the same as the package (ie, `<software>-<version>-<release>`), and copies the `README` file there. The newly created directory and the `README` file are marked in the RPM database as being documentation. The default documentation directory is `/usr/doc`, and can be changed by setting the `defaultdocdir` `rpmrc` file entry. For more information on `rpmrc` files, please see chapter B on page 363.

The file `/usr/local/foonly/README` was installed into that directory during the build and is included in the package file. When the package is installed, the `README` file is copied into `/usr/local/foonly` and marked in the RPM database as being documentation.

The `%config` Directive

The `%config` directive is used to flag the specified file as being a configuration file. RPM performs additional processing for config files when packages are erased, and during installations and upgrades. This is due to the nature of config files: They are often changed by the system administrator, and those changes should not be lost.

There is a restriction to the `%config` directive, and that restriction is that no more than one filename may follow the `%config`. This means that the following example is the only allowable way to specify config files:

```
%config /etc/foonly
```

Note that the full path to the file, as it is installed at build time, is required.

The `%attr` Directive

The `%attr` directive permits finer control over three key file attributes:

1. The file's permissions, or "mode".
2. The file's user ID.
3. The file's group ID.

The `%attr` directive has the following format:

```
%attr(<mode>, <user>, <group>) file
```

The mode is specified in the traditional numeric format, while the user and group are specifed as a string, such as "`root`". Here's a sample `%attr` directive:

```
%attr(755, root, root) foo.bar
```

This would set `foo.bar`'s permissions to 755. The file would be owned by user root, group root. If a particular attribute does not need to be specified (usually because the file is installed with that attribute set properly), then that attribute may be replaced with a dash:

```
%attr(755, -, root) foo.bar
```

The main reason to use the `%attr` directive is to permit users without root access to build packages. The techniques for doing this (and a more in-depth discussion of the `%attr` directive) can be found in chapter 16, on page 229.

The `%verify` Directive

RPM's ability to verify the integrity of the software it has installed is impressive. But sometimes it's a bit *too* impressive. After all, RPM can verify as many as nine different aspects of every file. The `%verify` directive can control which of these file attributes are to be checked when an RPM verification is done. Here are the attributes, along with the names used by the `%verify` directive:

1. Owner (`owner`)
2. Group (`group`)

3. Mode (`mode`)
4. MD5 Checksum (`md5`)
5. Size (`size`)
6. Major Number (`maj`)
7. Minor Number (`min`)
8. Symbolic Link String (`symlink`)
9. Modification Time (`mtime`)

How is `%verify` used? Say, for instance, that a package installs device files. Since the owner of a device will change, it doesn't make sense to have RPM verify the device file's owner/group and give out a false alarm. Instead, the following `%verify` directive could be used:

```
%verify(mode md5 size maj min symlink mtime) /dev/ttyS0
```

We've left out `owner` and `group`, since we'd rather RPM not verify those.[7] However, if all you want to do is prevent RPM from verifying one or two attributes, you can use `%verify`'s alternate syntax:

```
%verify(not owner group) /dev/ttyS0
```

This use of `%verify` produces identical results to the previous example.

13.6.2 Directory-related Directives

While the two directives in this section perform different functions, each is related to directories in some way. Let's see what they do:

The `%docdir` Directive

The `%docdir` directive is used to add a directory to the list of directories that will contain documentation. RPM includes the directories `/usr/doc`, `/usr/info`, and `/usr/man` in the `%docdir` list by default.

For example, if the following line is part of the `%files` list:

```
%docdir /usr/blather
```

any files in the `%files` list that RPM packages from `/usr/blather` will be included in the package as usual, but will also be automatically flagged as documentation. This directive is handy when a package creates its own documentation directory and contains a large number of files. Let's give it a try by adding the following line to our spec file:

[7] RPM will automatically exclude file attributes from verification if it doesn't make sense for the type of file. In our example, getting the MD5 checksum of a device file is an example of such a situation.

```
%docdir /usr/blather
```

Our `%files` list contains no references to the several files the package installs in the `/usr/blather` directory. After building the package, looking at the package's file list shows:

```
# rpm -qlp ../RPMS/i386/blather-1.0-1.i386.rpm
...
#
```

Wait a minute: There's nothing there, not even `/usr/blather`! What happened?

The problem is that `%docdir` only directs RPM to mark the specified directory as holding documentation. It *doesn't* direct RPM to package any files in the directory. To do that, we need to clue RPM in to the fact that there are files in the directory that must be packaged.

One way to do this is to simply add the files to the `%files` list:

```
%docdir /usr/blather
/usr/blather/INSTALL
```

Looking at the package, we see that `INSTALL` was packaged:

```
# rpm -qlp ../RPMS/i386/blather-1.0-1.i386.rpm
...
/usr/blather/INSTALL
#
```

Directing RPM to only show the documentation files, we see that `INSTALL` has indeed been marked as documentation, even though the `%doc` directive had not been used:

```
# rpm -qdp ../RPMS/i386/blather-1.0-1.i386.rpm
...
/usr/blather/INSTALL
#
```

Of course, if you go to the trouble of adding each file to the `%files` list, it wouldn't be that much more work to add `%doc` to each one. So the way to get the most benefit from `%docdir` is to add another line to the `%files` list:

```
%docdir /usr/blather
/usr/blather
```

Since the first line directs RPM to flag any file in `/usr/blather` as being documentation, and the second line tells RPM to automatically package any files found in `/usr/blather`, every single file in there will be packaged *and* marked as documentation:

```
# rpm -qdp ../RPMS/i386/blather-1.0-1.i386.rpm
/usr/blather
/usr/blather/COPYING
/usr/blather/INSTALL
/usr/blather/README
...
#
```

The `%docdir` directive can save quite a bit of effort in creating the `%files` list. The only caveat is that you must be sure the directory will only contain files you want marked as documentation. Keep in mind, also, that all subdirectories of the `%docdir`'ed directory will be marked as documentation directories, too.

The `%dir` Directive

As we mentioned in section 13.5, if a directory is specified in the `%files` list, the contents of that directory, and the contents of every directory under it, will automatically be included in the package. While this feature can be handy (assuming you are *sure* that every file under the directory should be packaged) there are times when this could be a problem.

The way to get around this, is to use the `%dir` directive. By adding this directive to the line containing the directory, RPM will package only the directory itself, regardless of what files are in the directory at the time the package is created. Here's an example of `%dir` in action.

The `blather-1.0` package creates the directory `/usr/blather` as part of its build. It also puts several files in that directory. In the spec file, the `/usr/blather` directory is included in the `%files` list:

```
%files
...
/usr/blather
...
```

There are no other entries in the `%files` list that have `/usr/blather` as part of their path. After building the package, we use RPM to look at the files in the package:

```
# rpm -qlp ../RPMS/i386/blather-1.0-1.i386.rpm
...
/usr/blather
/usr/blather/COPYING
/usr/blather/INSTALL
/usr/blather/README
...
#
```

The files present in /usr/blather at the time the package was built were included in the package automatically, without entering their names in the %files list.

However, after changing the /usr/blather line in the %files list to:

```
%dir /usr/blather
```

and rebuilding the package, a listing of the package's files now includes only the /usr/blather directory:

```
# rpm -qlp ../RPMS/i386/blather-1.0-1.i386.rpm
...
/usr/blather
...
#
```

-f *<file>* — Read the %files List From *<file>*

The -f option is used to direct RPM to read the %files list from the named file. Like the %files list in a spec file, the file named using the -f option should contain one filename per line and also include any of the directives named in this section.

Why is it necessary to read filenames from a file rather than have the filenames in the spec file? Here's a possible reason:

The filenames' paths may contain a directory name that can only be determined at build-time, such as an architecture specification. The list of files, minus the variable part of the path, can be created, and sed can be used at build-time to update the path appropriately.

It's not necessary that every filename to be packaged reside in the file. If there are any filenames present in the spec file, they will be packaged as well:

```
%files latex -f tetex-latex-skel
/usr/bin/latex
/usr/bin/pslatex
...
```

Here, the filenames present in the file tetex-latex-skel would be packaged, followed by every filename following the %files line.

13.7 The Lone Directive: %package

While every directive we've seen so far is used in the %files list, the %package directive is different. It is used to permit the creation of more than one package per spec file and can appear at any point in the spec file. These additional packages are known as subpackages. Subpackages are named according to the contents of the line containing the %package directive. The format of the package directive is:

```
%package: <string>
```

The <string> should be a name that describes the subpackage. This string is appended to the base package name to produce the subpackage's name. For example, if a spec file contains a `name` tag value of "`foonly`", and a "`%package doc`" line, then the subpackage name will be `foonly-doc`.

13.7.1 -n *<string>* — Use *<string>* As the Entire Subpackage Name

As we mentioned above, the name of a subpackage normally includes the main package name. When the `-n` option is added to the `%package` directive, it directs RPM to use the name specified on the `%package` line as the entire package name. In the example above, the following `%package` line would create a subpackage named `foonly-doc`:

```
%package doc
```

The following `%package` line would create a subpackage named `doc`:

```
%package -n doc
```

The `%package` directive plays another role in subpackage building. That role is to act as a place to collect tags that are specific to a given subpackage. Any tag placed after a `%package` directive will only apply to that subpackage.

Finally, the name string specified by the `%package` directive is also used to denote which parts of the spec file are a part of that subpackage. This is done by including the string (along with the `-n` option, if present on the `%package` line) on the starting line of the section that is to be subpackage-specific. Here's an example:

```
...
%package -n bar
...
%post -n bar
...
```

In this heavily edited spec file segment, a subpackage called `bar` has been defined. Later in the file is a post-install script. Because it has subpackage `bar`'s name on the `%post` line, the post-install script will be part of the `bar` subpackage only.

For more information on building subpackages, please see chapter 18 on page 251.

13.8 Conditionals

While the "exclude" and "exclusive" tags (`excludearch`, `exclusivearch`, `excludeos`, and `exclusiveos`) provide some control over whether a package will be built on a given architecture and/or operating system, that control is still rather coarse.

For example, what should be done if a package will build under multiple architectures, but requires slightly different `%build` scripts? Or what if a package requires a certain set of files under one operating system, and an entirely different set under another operating system? The architecture and operating system-specific tags we've discussed earlier in the chapter do nothing to help in such situations. What can be done?

One approach would be to simply create different spec files for each architecture or operating system. While it would certainly work, this approach has some problems:

- More work. The existence of multiple spec files for a given package means that the effort required to make any changes to the package is multiplied by however many different spec files there are.

- More chance for mistakes. If any work needs to be done to the spec files, the fact they are separate means it is that much easier to forget to make the necessary changes to each one. There is also the chance of introducing mistakes each time changes are made.

The other approach is to somehow permit the conditional inclusion of architecture- or operating system-specific sections of the spec file. Fortunately, the RPM designers chose this approach, and it makes multi-platform package building easier and less prone to mistakes.

We discuss multi-platform package building in depth in chapter 19. For now, let's take a quick look at RPM's conditionals.

The `%ifarch` Conditional

The `%ifarch` conditional is used to begin a section of the spec file that is architecture-specific. It is followed by one or more architecture specifiers, each separated by commas or whitespace. Here is an example:

```
%ifarch i386 sparc
```

The contents of the spec file following this line would be processed only by Intel x86 or Sun SPARC-based systems. However, if only this line were placed in a spec file, this is what would happen if a build was attempted:

```
# rpm -ba cdplayer-1.0.spec
Unclosed %if
Build failed.
#
```

The problem that surfaced here is that any conditional must be "closed" by using either `%else` or `%endif`. We'll be covering them a bit later in the chapter.

The `%ifnarch` Conditional

The `%ifnarch` conditional is used in a similar fashion to `%ifarch`, except that the logic is reversed. If a spec file contains a conditional block starting with `%ifarch alpha`, that block would be processed only if the build was being done on a Digital Alpha/AXP-based system. However, if the conditional block started with `%ifnarch alpha`, then that block would be processed only if the build were *not* being done on an Alpha.

Like `%ifarch`, `%ifnarch` can be followed by one or more architectures and must be closed by a `%else` or `%endif`.

The `%ifos` Conditional

The `%ifos` conditional is used to control RPM's spec file processing based on the build system's operating system. It is followed by one or more operating system names. A conditional block started with `%ifos` must be closed by a `%else` or `%endif`. Here's an example:

```
%ifos linux
```

The contents of the spec file following this line would be processed only if the build was done on a linux system.

The `%ifnos` Conditional

The `%ifnos` conditional is the logical complement to `%ifos`: that is, if a conditional starting with the line `%ifnos irix` is present in a spec file, then the file contents after the `%ifnos` will not be processed if the build system is running Irix. As always, a conditional block starting with `%ifnos` must be closed by a `%else` or `%endif`.

The `%else` Conditional

The `%else` conditional is placed between a `%if` conditional of some persuasion, and a `%endif`. It is used to create two blocks of spec file statements, only one of which will be used in any given case. Here's an example:

```
%ifarch alpha
make RPM_OPT_FLAGS="$RPM_OPT_FLAGS -I ."
%else
make RPM_OPT_FLAGS="$RPM_OPT_FLAGS"
%endif
```

When a build is performed on a Digital Alpha/AXP, some additional flags are added to the `make` command. On all other systems, these flags are not added.

The %endif Conditional

A %endif is used to end a conditional block of spec file statements. It can follow one of the %if conditionals, or the %else. The %endif is always needed after a conditional, otherwise the build will fail. Here's short conditional block, ending with a %endif:

```
%ifarch i386
make INTELFLAG=-DINTEL
%endif
```

In this example, we see the conditional block started with a %ifarch and ended with a %endif.

Now that we have some more in-depth knowledge of the spec file, let's take a look at some of RPM's additional features. In the next chapter, we'll explore how to add dependency information to a package.

Chapter 14

Adding Dependency Information to a Package

Since the very first version of RPM hit the streets, one of the side effects of RPM's ease of use was that it made it easier for people to break things. Since RPM made it so simple to erase packages, it became common for people to joyfully erase packages until something broke.

Usually this only bit people once, but even once was too much of a hassle if it could be prevented. With this in mind, the RPM developers gave RPM the ability to:

- Build packages that contain information on the capabilities they require.
- Build packages that contain information on the capabilities they provide.
- Store this "provides" and "requires" information in the RPM database.

In addition, they made sure RPM was able to display dependency information, as well as to warn users if they were attempting to do something that would break a package's dependency requirements.

With these features in place, it became more difficult for someone to unknowingly erase a package and wreak havoc on their system.

14.1 An Overview of Dependencies

We've already alluded to the underlying concept for RPM's dependency processing. It is based on two key factors:

- Packages advertise what capabilities they provide.
- Packages advertise what capabilities they require.

By simply checking these two types of information, many possible problems can be avoided. For example, if a package requires a capability that is not provided by any already-installed package, that package cannot be installed and expected to work properly.

On the other hand, if a package is to be erased, but its capabilities are required by other installed packages, then it cannot be erased without causing other packages to fail.

As you might imagine, it's not quite *that* simple. But adding dependency information can be easy. In fact, in most cases, it's automatic!

14.2 Automatic Dependencies

When a package is built by RPM, if any file in the package's `%files` list is a shared library, the library's "soname" is automatically added to the list of capabilities the package provides. The soname is the name used to determine compatibility between different versions of a library.

Note that this is *not* a filename. In fact, no aspect of RPM's dependency processing is based on filenames. Many people new to RPM often make the assumption that a failed dependency represents a missing file. This is not the case.

Remember that RPM's dependency processing is based on knowing what capabilities are provided by a package and what capabilities a package requires. We've seen how RPM automatically determines what shared library resources a package provides. But does it automatically determine what shared libraries a package *requires*?

Yes! RPM does this by running `ldd` on every executable program in a package's `%files` list. Since `ldd` provides a list of the shared libraries each program requires, both halves of the equation are complete — that is, the packages that make shared libraries available, *and* the packages that require those shared libraries, are tracked by RPM. RPM can then take that information into account when packages are installed, upgraded, or erased.

14.2.1 The Automatic Dependency Scripts

RPM uses two scripts to handle automatic dependency processing. They reside in `/usr/bin` and are called `find-requires`, and `find-provides`. We'll take a look at them in a minute, but first let's look at why there are scripts to do this sort of thing. Wouldn't it be better to have this built into RPM itself?

Actually, creating scripts for this sort of thing *is* a better idea. The reason? RPM has already been ported to a variety of different operating systems. Determining what shared libraries an executable requires, and the soname of shared libraries, is simple, but the exact steps required vary widely from one operating system to another. Putting this part of RPM into a script makes it easier to port RPM.

Let's take a look at the scripts that are used by RPM under the Linux operating system.

`find-requires` — **Automatically Determine Shared Library Requirements**

The `find-requires` script for Linux is quite simple:

```
#!/bin/sh

# note this works for both a.out and ELF executables

ulimit -c 0

filelist=`xargs -r file | fgrep executable | cut -d: -f1 `

for f in $filelist; do
    ldd $f | awk '/=>/ { print $1 }'
done | sort -u | xargs -r -n 1 basename | sort -u
```

This script first creates a list of executable files. Then, for each file in the list, `ldd` determines the file's shared library requirements, producing a list of sonames. Finally, the list of sonames is sanitized by removing duplicates, and removing any paths.

`find-provides` — **Automatically Determine Shared Library Sonames**

The `find-provides` script for Linux is a bit more complex, but still pretty straightforward:

```
#!/bin/bash

# This script reads filenames from STDIN and outputs any relevant
# provides information that needs to be included in the package.

filelist=$(grep "\\.so" | grep -v "^/lib/ld.so" |
xargs file -L 2>/dev/null | grep "ELF.*shared object" | cut -d: -f1)

for f in $filelist; do
    soname=$(objdump -p $f | awk '/SONAME/ {print $2}')

    if [ "$soname" != "" ]; then
        if [ ! -L $f ]; then
            echo $soname
        fi
    else
        echo ${f##*/}
    fi
done | sort -u
```

First, a list of shared libraries is created. Then, for each file on the list, the soname is extracted, cleaned up, and duplicates removed.

14.2.2 Automatic Dependencies: An Example

Let's take a widely used program, `ls`, the directory lister, as an example. On a Red Hat Linux system, `ls` is part of the `fileutils` package and is installed in `/bin`. Let's play the part of RPM during `fileutils`' package build and run `find-requires` on `/bin/ls`. Here's what we'll see:

```
# find-requires
/bin/ls
<ctrl-d>
libc.so.5
#
```

The `find-requires` script returned `libc.so.5`. Therefore, RPM should add a requirement for `libc.so.5` when the `fileutils` package is built. We can verify that RPM did add `ls`' requirement for `libc.so.5` by using RPM's `--requires` option to display `fileutils`' requirements:

```
# rpm -q --requires fileutils
libc.so.5
#
```

OK, that's the first half of the equation — RPM automatically detecting a package's shared library requirements. Now let's look at the second half of the equation – RPM detecting packages that provide shared libraries. Since the `libc` package includes, among others, the shared library `/lib/libc.so.5.3.12`, RPM would obtain its soname. We can simulate this by using `find-provides` to print out the library's soname:

```
# find-provides
/lib/libc.so.5.3.12
<ctrl-d>
libc.so.5
#
```

OK, so `/lib/libc.so.5.3.12`'s soname is `libc.so.5`. Let's see if the `libc` package really does "provide" the `libc.so.5` soname:

```
# rpm -q --provides libc
libm.so.5
libc.so.5
#
```

Yes, there it is, along with the soname of another library contained in the package. In this way, RPM can ensure that any package requiring `libc.so.5` will have a compatible library available as long as the `libc` package, which provides `libc.so.5`, is installed.

In most cases, automatic dependencies are enough to fill the bill. However, there are circumstances when the package builder has to manually add dependency information to a package. Fortunately, RPM's approach to manual dependencies is both simple and flexible.

14.2.3 The `autoreqprov` Tag — Disable Automatic Dependency Processing

There may be times when RPM's automatic dependency processing is not desired. In these cases, the `autoreqprov` tag may be used to disable it. This tag takes a `yes/no` or `0/1` value. For example, to disable automatic dependency processing, the following line may be used:

```
AutoReqProv: no
```

14.3 Manual Dependencies

You might have noticed that we've been using the words "requires" and "provides" to describe the dependency relationships between packages. As it turns out, these are the exact words used in spec files to manually add dependency information. Let's look at the first tag: `requires`.

14.3.1 The `requires` Tag

We've been deliberately vague when discussing exactly what it is that a package requires. Although we've used the word "capabilities", in fact, manual dependency requirements are always represented in terms of packages. For example, if package `foo` requires that package `bar` is installed, it's only necessary to add the following line to `foo`'s spec file:

```
requires: bar
```

Later, when the `foo` package is being installed, RPM will consider `foo`'s dependency requirements met if any version of package `bar` is already installed.[1]

If more than one package is required, they can be added to the `requires` tag, one after another, separated by commas and/or spaces. So if package `foo` requires packages `bar` *and* `baz`, the following line will do the trick:

```
requires: bar, baz
```

[1]As long as the requiring *and* the providing packages are installed using the same invocation of RPM, the dependency checking will succeed. For example, the command `rpm -ivh *.rpm` will properly check for dependencies, even if the requiring package ends up being installed *before* the providing package.

As long as any version of `bar` and `baz` is installed, `foo`'s dependencies will be met.

Adding Version Requirements

When a package has slightly more stringent needs, it's possible to require certain versions of a package. All that's necessary is to add the desired version number, preceded by one of the following comparison operators:

- `<` Requires package with a version less than the specified version.
- `<=` Requires package with a version less than or equal to the specified version.
- `=` Requires package with a version equal to the specified version.
- `>=` Requires package with a version equal to or greater than the specified version.
- `>` Requires package with a version greater than the specified version.

Continuing with our example, let's suppose that the required version of package `bar` actually needs to be at least 2.7, and that the `baz` package must be version 2.1 — no other version will do. Here's what the `requires` tag line would look like:

```
requires:  bar >= 2.7, baz = 2.1
```

We can get even more specific and require a particular *release* of a package:

```
requires:  bar >= 2.7-4, baz = 2.1-1
```

When Version Numbers Aren't Enough

You might think that with all these features, RPM's dependency processing can handle every conceivable situation. You'd be right, except for the problem of version numbers. RPM needs to be able to determine which version numbers are more recent than others, in order to perform its version comparisons.

It's pretty simple to determine that version 1.5 is older than version 1.6. But what about 2.01 and 2.1? Or 7.6a and 7.6? There's no way for RPM to keep up with all the different version-numbering schemes in use. But there *is* a solution; two, in fact...

Solution Number 1: Serial numbers When RPM can't decipher a package's version number, it's time to pull out the `serial` tag. This tag is used to help RPM determine version number ordering. Here's a sample `serial` tag line:

```
Serial:  42
```

This line indicates that the package has a serial number of 42. What does the 42 mean? Only that this version of the package is older than the same package with a serial number of 41, but younger than the same package with a serial number of 43. If you think of serial numbers as being nothing more than very simple version numbers, you'll be on the mark.

In order to direct RPM to look at the serial number instead of the version number when doing dependency checking, it's necessary to append an "`S`" to the end of the conditional operator in the `requires` tag line. So if a package requires package `foo` to have a serial number equal to 42, the following tag line would be used:

```
Requires:  foo =S 42
```

If the `foo` package needs to have a serial number greater than or equal to 42, this line would work:

```
Requires:  foo >=S 42
```

It might seem that using serial numbers is a lot of extra trouble, and you're right. But there is an alternative:

Solution Number 2: Just Say No! If you have the option between changing the software's version-numbering scheme, or using serial numbers in RPM, please consider changing the version-numbering scheme. Chances are, if RPM can't figure it out, most of the people using your software can't, either. But in case you aren't the author of the software you're packaging, and its version numbering scheme is giving RPM fits, the `serial` tag can help you out.

14.3.2 The `conflicts` Tag

The `conflicts` tag is the logical complement to the `requires` tag. It is used to specify which packages conflict with the current package. RPM will not permit conflicting packages to be installed unless overridden with the `--nodeps` option.

The `conflicts` tag has the same format as `requires`. It accepts a real or virtual package name and can optionally include version and release specifications or a serial number.

14.3.3 The `provides` Tag

Now that you've seen how it's possible to require a package using the `requires` tag, you're probably expecting that you'll need to use the `provides` tag in every single package. After all, RPM has to get those package names from *somewhere*, right?

While it is true that RPM needs to have the package names available, the `provides` tag is normally not required. It would actually be redundant, because the RPM database already contains the name of every package installed. There's no need to duplicate that information.

But wait – We said earlier that manual dependency requirements are *always* represented in terms of packages. If RPM doesn't require the package builder to use the `provides` tag to provide the package name, then what is the `provides` tag used for?

Virtual Packages

Enter the virtual package. A virtual package is nothing more than a name specified with the `provides` tag. Virtual packages are handy when a package requires a certain capability, and that capability can be provided by any one of several packages. Here's an example:

In order to work properly, `sendmail` needs a local delivery agent to handle mail delivery. There are a number of different local delivery agents available — `sendmail` will work just fine with any of them.

In this case, it doesn't make sense to force the use of a particular local delivery agent; as long as one's installed, `sendmail`'s requirements will have been satisfied. So `sendmail`'s package builder adds the following line to `sendmail`'s spec file:

```
requires:  lda
```

There is no package with that name available, so `sendmail`'s requirements must be met with a virtual package. The creators of the various local delivery agents indicate that their packages satisfy the requirements of the `lda` virtual package by adding the following line to *their* packages' spec files:

```
provides:  lda
```

(Note that virtual packages may not have version numbers.) Now, when `sendmail` is installed, as long as there is a package installed that provides the `lda` virtual package, there will be no problem.

14.4 To Summarize...

RPM's dependency processing is based on tracking the capabilities a package provides, and the capabilities a package requires. A package's requirements can come from two places:

1. Shared library requirements, automatically determined by RPM.
2. The `requires` tag line, manually added to the package's spec file.

These requirements can be viewed by using RPM's `--requires` query option. A specific requirement can be viewed by using the `--whatrequires` query option. Both options are fully described in chapter 5.

The capabilities a package provides, can come from three places:

1. Shared library sonames, automatically determined by RPM.
2. The `provides` tag line, manually added to the package's spec file.
3. The package's name (and optionally, version/serial number).

The first two types of information can be viewed by using RPM's `--provides` query option. A specific capability can be viewed by using the `--whatprovides` query option. Both options are fully described in chapter 5.

The package name and version are not considered capabilities that are explicitly provided. Therefore, if a search using `--provides` or `--whatprovides` comes up dry, try simply looking for a package by that name.

As you've probably gathered by now, using manual dependencies requires some level of synchronization between packages. This can be tricky, particularly if you're not responsible for both packages. But RPM's dependency processing can make life easier for your users.

Chapter 15

Making a Relocatable Package

RPM has the ability to give users some latitude in deciding where packages are to be installed on their systems. However, package builders must first design their packages to give users this freedom.

That's all well and good, but why would the ability to "relocate" a package be all that important?

15.1 Why relocatable packages?

One of the many problems that plague a system administrator's life is disk space. Usually, there's not enough of it, and if there *is* enough, chances are it's in the wrong place. Here's a hypothetical example:

- Some new software comes out and is desired greatly by the user community.
- The system administrator carefully reviews the software's installation documentation prior to doing to the installation.[1] She notes that the software, all 150MB of it, installs into `/opt`.
- Frowning, the sysadmin fires off a quick `df` command:

  ```
  # df
  Filesystem         1024-blocks  Used Available Capacity Mounted on
  /dev/sda0             100118   28434    66514     30%   /
  /dev/sda6             991995  365527   575218     39%   /usr
  #
  ```

  Bottom line: There's no way 150MB of new software is going to fit on the root filesystem.

[1] Hey, we said it was hypothetical!

- Sighing heavily, the sysadmin ponders what to do next. If only there were some way to install the software somewhere on the `/usr` filesystem...

It doesn't have to be this way. RPM has the ability to make packages that can be installed with a user-specified prefix that dictates where the software will actually be placed. By making packages relocatable, the package builder can make life easier for sysadmins everywhere. But what exactly *is* a relocatable package?

A relocatable package is a package that is standard in every way, save one. The difference lies in the `prefix` tag. When this tag is added to a spec file, RPM will attempt to build a relocatable package.

Note the word "attempt". There are a few conditions that must be met before a relocatable package can be built successfully, and this chapter will cover them all. But first, let's look at exactly how RPM can relocate a package. And the one component at the heart of package relocation is the `prefix` tag.

15.2 The `prefix` tag: Relocation Central

The best way to explain how the `prefix` tag is used is to step through an example. Here's a sample `prefix` tag:

```
Prefix: /opt
```

In this example, the prefix path is defined as `/opt`. This means that, by default, the package will install its files under `/opt`. Let's assume the spec file contains the following line in its `%files` list:

```
/opt/bin/baz
```

If the package is installed without any relocation, this file will be installed in `/opt/bin`. This is identical to how a non-relocatable package is installed.

However, if the package *is* to be relocated on installation, the path of every file in the `%files` list is modified according to the following steps:

1. The part of the file's path that corresponds to the path specified on the `prefix` tag line is removed.
2. The user-specified relocation prefix is prepended to the file's path.

Using our `/opt/bin/baz` file as an example, let's assume that the user installing the package wishes to override the default prefix (`/opt`), with a new prefix, say, `/usr/local/opt`. Following the steps above, we first remove the original prefix from the file's path:

`/opt/bin/baz` becomes `/bin/baz`

Next, we add the user-specified prefix to the front of the remaining part of the filename:

/usr/local/opt + /bin/baz = /usr/local/opt/bin/baz

Now that the file's new path has been created, RPM installs the file normally. This part of it seems simple enough, and it is. But as we mentioned above, there are a few things the package builder needs to consider before getting on the relocatable package bandwagon.

15.3 Relocatable Wrinkles: Things to Consider

While it's certainly no problem to add a `prefix` tag line to a spec file, it's necessary to consider a few other issues:

- Every file in the `%files` list must start with the path specified on the `prefix` tag line.
- The software must be written such that it can operate properly if relocated. Absolute symlinks are a prime example of this.
- Other software must not rely on the relocatable package being installed in any particular location.
- inside-install/erase

Let's cover each of these issues, starting with the `%files` list.

15.3.1 `%files` List Restrictions

As mentioned above, each file in the `%files` list must start with the relocation prefix. If this isn't done, the build will fail:

```
# rpm -ba cdplayer-1.0.spec
* Package: cdplayer
+ umask 022
+ echo Executing: %prep
...
Binary Packaging: cdplayer-1.0-1
Package Prefix = usr/local
File doesn't match prefix (usr/local): /usr/doc/cdplayer-1.0-1
File not found: /usr/doc/cdplayer-1.0-1
Build failed.
#
```

In our example, the build proceeded normally until the time came to create the binary package file. At that point RPM detected the problem. The error message says it all: The `prefix` line in the spec file (`/usr/local`) was not present in the first part of the file's (`/usr/doc/cdplayer-1.0-1`) path. This stopped the build in its tracks.

The fact that every file in a relocatable package must be installed under the directory specified in the `prefix` line, raises some issues. For example, what about a program that reads a configuration file normally kept in `/etc`? This question leads right into our next section.

15.3.2 Relocatable Packages Must Contain Relocatable Software

While this section's title seems pretty obvious, it's not always easy to tell if a particular piece of software can be relocated. Let's take a look at the question raised at the end of the previous section. If a program has been written to read its configuration file from `/etc`, there are three possible approaches to making that program relocatable:

1. Set the prefix to `/etc` and package everything under `/etc`.
2. Package everything somewhere other than `/etc` and leave out the config file entirely.
3. Modify the program.

The first approach would certainly work from a purely technical standpoint, but not many people would be happy with a program that installed itself in `/etc`. So this approach isn't viable.

The second approach might be more appropriate, but it forces users to complete the install by having them create the config file themselves. If RPM's goal is to make software easier to install and remove, this is not a viable approach, either!

The final approach might be the best. Once the program is installed, when the rewritten software is first run, it could see that no configuration file existed in `/etc`, and create one. However, even though this would work, when the time came to erase the package, the config file would be left behind. RPM had never installed it, so RPM couldn't get rid of it. There's also the fact that this approach is probably more labor intensive than most package builders would like.

None of these approaches are very appealing, are they? Some software just doesn't relocate very well. In general, any of the following things are warning signs that relocation is going to be a problem:

- The software contains one or more files that must be installed in specific directories
- The software refers to system files using relative paths (Which is really just another way of saying the software must be installed in a particular directory)

If these kinds of issues crop up, then making the software relocatable is going to be tough. And there's still one issue left to consider.

15.3.3 The Relocatable Software Is Referenced By Other Software

Even assuming the software is written so that it can be put in a relocatable package, there still might be a problem. And that problem centers not on the relocatable software itself, but on other programs that reference the relocatable software.

For example, there are times when a package needs to execute other programs. This might include backup software that needs to send mail, or a communications program that needs to compress files. If these underlying programs were relocatable, and not installed where other packages expect them, then they would be of little use.

Granted, this isn't a common problem, but it can happen. And for the package builder interested in building relocatable packages, it's an issue that needs to be explored. Unfortunately, this type of problem can be the hardest to find.

If, however, a software product has been found to be relocatable, the mechanics of actually building a relocatable package are pretty straightforward. Let's give it a try.

15.4 Building a Relocatable Package

For this example, we'll use our tried-and-true `cdplayer` application. Let's start by reviewing the spec file for possible problems:

```
#
# Example spec file for cdplayer app...
#
Summary:A CD player app that rocks!
Name: cdplayer
...
%files
%doc README
/usr/local/bin/cdp
/usr/local/bin/cdplay
%doc /usr/local/man/man1/cdp.1
%config /etc/cdp-config
```

Everything looks all right, except for the `%files` list. There are files in `/usr/local/bin`, a `man` page in `/usr/local/man/man1`, and a config file in `/etc`. A prefix of `/usr/local` would work pretty well, except for that `cdp-config` file.

For the sake of this first build, we'll declare the config file unnecessary and remove it from the `%files` list. We'll then add a `prefix` tag line, setting the prefix to `/usr/local`. After these changes are made, let's try a build:

```
# rpm -ba cdplayer-1.0.spec
* Package: cdplayer
+ umask 022
+ echo Executing: %prep
Executing: %prep
+ cd /usr/src/redhat/BUILD
+ cd /usr/src/redhat/BUILD
+ rm -rf cdplayer-1.0
+ gzip -dc /usr/src/redhat/SOURCES/cdplayer-1.0.tgz
```

```
...
Binary Packaging: cdplayer-1.0-1
Package Prefix = usr/local
File doesn't match prefix (usr/local): /usr/doc/cdplayer-1.0-1
File not found: /usr/doc/cdplayer-1.0-1
Build failed.
#
```

The build proceeded normally up to the point of actually creating the binary package. The `Package Prefix = usr/local` line confirms that RPM picked up our `prefix` tag line. But the build stopped – and on a file called `/usr/doc/cdplayer-1.0-1`. But that file isn't even in the `%files` list. What's going on?

Take a closer look at the `%files` list. See the line that reads `%doc README`? In section 13.6.1 on page 199, we discussed how the `%doc` directive creates a directory under `/usr/doc`. That's the file that killed the build – the directory created by the `%doc` directive.

Let's temporarily remove that line from the `%files` list and try again:

```
# rpm -ba cdplayer-1.0.spec
* Package: cdplayer
+ umask 022
+ echo Executing: %prep
Executing: %prep
+ cd /usr/src/redhat/BUILD
+ cd /usr/src/redhat/BUILD
+ rm -rf cdplayer-1.0
+ gzip -dc /usr/src/redhat/SOURCES/cdplayer-1.0.tgz
...
Binary Packaging: cdplayer-1.0-1
Package Prefix = usr/local
Finding dependencies...
Requires (2): libc.so.5 libncurses.so.2.0
bin/cdp
bin/cdplay
man/man1/cdp.1
90 blocks
Generating signature: 0
Wrote: /usr/src/redhat/RPMS/i386/cdplayer-1.0-1.i386.rpm
+ umask 022
+ echo Executing: %clean
Executing: %clean
+ cd /usr/src/redhat/BUILD
+ cd cdplayer-1.0
+ exit 0
Source Packaging: cdplayer-1.0-1
cdplayer-1.0.spec
cdplayer-1.0.tgz
```

```
82 blocks
Generating signature: 0
Wrote: /usr/src/redhat/SRPMS/cdplayer-1.0-1.src.rpm
#
```

The build completed normally. Note how the files to be placed in the binary package file are listed, minus the prefix of `/usr/local`. Some of you might be wondering why the `cdp.1` file didn't cause problems. After all, it had a `%doc` directive, too. The answer lies in the way the file was specified. Since the file was specified using an absolute path, and that path started with the prefix `/usr/local`, there was no problem. A more complete discussion of the `%doc` directive can be found in section 13.6.1 on page 199.

15.4.1 Tying Up the Loose Ends

In the course of building this package, we ran into two hitches:

1. The config file `cdp-config` couldn't be installed in `/etc`.

2. The `README` file could not be packaged using the `%doc` directive.

Both of these issues are due to the fact that the files' paths do not start with the default prefix path `/usr/local`. Does this mean this package cannot be relocated? Possibly, but there are two options to consider. The first option is to review the prefix. In the case of our example, if we chose a prefix of `/usr` instead of `/usr/local`, the `README` file could be packaged using the `%doc` directive, since the default documentation directory is `/usr/doc`. Another approach would be to use the `%docdir` directive to define another documentation-holding directory somewhere along the prefix path.[2]

This approach wouldn't work for `/etc/cdp-config`, though. To package that file, we'd need to resort to more extreme measures. Basically, this approach would entail packaging the file in an acceptable directory (something under `/usr/local`) and using the `%post` post-install script to move the file to `/etc`. Pointing a symlink at the config file is another possibility.

Of course, this approach has some problems. First, you'll need to write a `%postun` script to undo what the `%post` script does.[3] A `%verifyscript` that made sure the files were in place would be nice, too. Second, because the file or symlink wasn't installed by RPM, there's no entry for it in the RPM database. This reduces the utility of RPM's `-c` and `-d` options when issuing queries. Finally, if you actually move files around using the `%post` script, the files you move will not verify properly, and when the package is erased, your users will get some disconcerting messages when RPM can't find the moved files to erase them. If you have to resort to these kinds of tricks, it's probably best to forget trying to make the package relocatable.

[2] For more information on the `%docdir` directive, please see section 13.6.2 on page 201.

[3] Install and erase-time scripts have an environment variable, `RPM_INSTALL_PREFIX`, that can be used to write scripts that are able to act appropriately if the package is relocated. See section 13.3.2 on page 183 for more information.

15.4.2 Test-Driving a Relocatable Package

Looks like cdplayer is a poor candidate for being made relocatable. However, since we did get a hamstrung version to build successfully, we can use it to show how to test a relocatable package.

First, let's see if the binary package file's prefix has been recorded properly. We can do this by using the --queryformat option to RPM's query mode:

```
# rpm -qp --queryformat '%{DEFAULTPREFIX}\n' cdplayer-1.0-1.i386.rpm
/usr/local
#
```

The DEFAULTPREFIX tag directs RPM to display the prefix used during the build. As we can see, it's /usr/local, just as we intended. The --queryformat option is discussed in chapter 5, specifically, in section 5.2.2 on page 67.

So it looks like we have a relocatable package. Let's try a couple of installs and see if we really *can* install it in different locations. First, let's try a regular install with no prefix specified:

```
# rpm -Uvh cdplayer-1.0-1.i386.rpm
cdplayer    ##################################################
#
```

That seemed to work well enough. Let's see if the files went where we intended:

```
# ls -al /usr/local/bin
total 558
-rwxr-xr-x 1 root root 40739 Oct 7 13:23 cdp*
lrwxrwxrwx 1 root root 18 Oct 7 13:40 cdplay -> /usr/local/bin/cdp*
...
# ls -al /usr/local/man/man1
total 9
-rwxr-xr-x 1 root root 4550 Oct 7 13:23 cdp.1*
...
#
```

Looks good. Let's erase the package and reinstall it with a different prefix:

```
# rpm -e cdplayer
# rpm -Uvh --prefix /usr/foonly/blather cdplayer-1.0-1.i386.rpm
cdplayer    ##################################################
#
```

(We should mention that directories `foonly` and `blather` didn't exist prior to installing `cdplayer`.)

RPM has another tag that can be used with the `--queryformat` option. It's called `INSTALLPREFIX` and using it displays the prefix under which a package was installed. Let's give it a try:

```
# rpm -q --queryformat '%{INSTALLPREFIX}\n' cdplayer
/usr/foonly/blather
#
```

As we can see, it displays the prefix we entered on the command line. Let's see if the files were installed as we directed:

```
# cd /usr/foonly/blather/
# ls -al
total 2
drwxr-xr-x 2 root root 1024 Oct 7 13:45 bin/
drwxr-xr-x 3 root root 1024 Oct 7 13:45 man/
#
```

So far, so good – the proper directories are there. Let's look at the `man` page first:

```
# cd /usr/foonly/blather/man/man1/
# ls -al
total 5
-rwxr-xr-x 1 root root 4550 Oct 7 13:23 cdp.1*
#
```

That looks ok. Now on to the files in `bin`:

```
# cd /usr/foonly/blather/bin
# ls -al
total 41
-rwxr-xr-x 1 root root 40739 Oct 7 13:23 cdp*
lrwxrwxrwx 1 root root 18 Oct 7 13:45 cdplay -> /usr/local/bin/cdp
#
```

Uh-oh. That `cdplay` symlink isn't right. What happened? If we look at `cdplayer`'s makefile, we see the answer:

```
install: cdp cdp.1.Z
...
ln -s /usr/local/bin/cdp /usr/local/bin/cdplay
```

Ah, when the software is installed during RPM's build process, the make file sets up the symbolic link. Looking back at the `%files` list, we see `cdplay` listed. RPM blindly packaged the symlink, complete with its non-relocatable string. This is why we mentioned absolute symlinks as a prime example of non-relocatable software.

Fortunately, this problem isn't that difficult to fix. All we need to do is change the line in the makefile that creates the symlink from:

```
ln -s /usr/local/bin/cdp /usr/local/bin/cdplay
```

To:

```
ln -s ./cdp /usr/local/bin/cdplay
```

Now `cdplay` will always point to `cdp`, no matter where it's installed. When building relocatable packages, relative symlinks are your friend!

After rebuilding the package, let's see if our modifications have the desired effect. First, a normal install with the default prefix:

```
# rpm -Uvh --nodeps cdplayer-1.0-1.i386.rpm
cdplayer    ##################################################

# cd /usr/local/bin/
# ls -al cdplay
lrwxrwxrwx 1 root root 18 Oct 8 22:32 cdplay -> ./cdp*
```

Next, we'll try a second install using the `--prefix` option (after we first delete the original package):

```
# rpm -e cdplayer
# rpm -Uvh --nodeps --prefix /a/dumb/prefix cdplayer-1.0-1.i386.rpm
cdplayer    ##################################################

# cd /a/dumb/prefix/bin/
# ls -al cdplay
lrwxrwxrwx 1 root root 30 Oct 8 22:34 cdplay -> ./cdp*
#
```

As you can see, the trickiest part about building relocatable packages is making sure the software you're packaging is up to the task. Once that part of the job is done, the actual modifications are straightforward.

In the next chapter, we'll cover how packages can be built in non-standard directories, as well as how non-root users can build packages.

Chapter 16

Making a Package That Can Build Anywhere

While RPM makes building packages as easy as possible, some of the default design decisions might not work well in a particular situation. Here are two situations where RPM's method of package building may cause problems:

1. You are unable to dedicate a system to RPM package building, or the software you're packaging would disrupt the build system's operation if it were installed.

2. You would like to package software, but you don't have root access to an appropriate build system.

Either of these situations can be resolved by directing RPM to build, install, and package the software in a different area on your build system. It requires a bit of additional effort to accomplish this, but taken a step at a time, it is not difficult. Basically, the process can be summed up by addressing the following steps:

- Writing the package's spec file to support a build root.

- Directing RPM to build software in a user-specified build area.

- Specifying file attributes that RPM needs to set on installation.

The methods discussed here are not required in every situation. For example, a system administrator developing a package on a production system may only need to add support for a build root. On the other hand, a student wishing to build a package on a university system will need to get around the lack of root access by implementing every method described here.

16.1 Using Build Roots in a Package

Part of the process of packaging software with RPM is to actually build the software and install it on the build system. The installation of software can only be accomplished by someone with root access, so a non-privileged user will certainly need to handle RPM's installation phase differently. There are times, however, when even a person with root access will not want RPM to copy new files into the system's directories. As mentioned above, the reasons might be due to the fact that the software being packaged is already in use on the build system. Another reason might be as mundane as not having enough free space available to perform the install into the default directories.

Whatever the reason, RPM provides the ability to direct a given package to install into an alternate root. This alternate root is known as a "build root". Several requirements must be met in order for a build root to be utilized:

- A default build root must be defined in the package's spec file.
- The installation method used by the software being packaged must be able to support installation in an alternate root.

The first part is easy. It entails adding the following line to the spec file:

```
BuildRoot: <root>
```

Of course, you would replace "`<root>`" with the name of the directory in which you'd like the software to install.[1] If, for example, you specify a build root of `/tmp/foo`, and the software you're packaging installs a file `bar` in `/usr/bin`, you'll find `bar` residing in `/tmp/foo/usr/bin` after the build.

A note for you non-root package builders: make sure you can actually write to the build root you specify! Those of you with root access should also make sure you choose your build root carefully. For an assortment of reasons, it's *not* a good idea to declare a build root of "/"! We'll get into the reasons why shortly.

The final requirement for adding build root support is to make sure the software's installation method can support installing into an alternate root. The difficulty in meeting this requirement can range from dead simple to nearly impossible. There are probably as many different ways of approaching this as there are packages to build. But in general, some variant of the following approach is used:

- The environment variable `RPM_BUILD_ROOT` is set by RPM and contains the value of the build root to be used when the software is built and installed.
- The `%install` section of the spec file is modified to use `RPM_BUILD_ROOT` as part of the installation process.

[1] Keep in mind that the build root can be overridden at build-time using the `--buildroot` option or the `buildroot` `rpmrc` file entry. See chapter 12 on page 139 for more details.

- If the software is installed using `make`, the makefile is modified to use `RPM_BUILD_ROOT` and to create any directories that may not exist at installation time.

Here's an example of how these components work together to utilize a build root. First, there's the definition of the build root in the spec file:

```
BuildRoot: /tmp/cdplayer
```

This line defines the build root as being `/tmp/cdplayer`. All the files installed by this software will be placed under the `cdplayer` directory. Next is the spec file's `%install` section:

```
%install
make ROOT="$RPM_BUILD_ROOT" install
```

Since the software we're packaging uses `make` to perform the actual install, we simply define the environment variable `ROOT` to be the path defined by `RPM_BUILD_ROOT`. So far, so good. Things really start to get interesting in the software's `Makefile`, though:

```
install: cdp cdp.1.Z
#       chmod 755 cdp
#       cp cdp /usr/local/bin
        install -m 755 -o 0 -g 0 -d $(ROOT)/usr/local/bin/
        install -m 755 -o 0 -g 0 cdp $(ROOT)/usr/local/bin/cdp
#       ln -s /usr/local/bin/cdp /usr/local/bin/cdplay
        ln -s ./cdp $(ROOT)/usr/local/bin/cdplay
#       cp cdp.1 /usr/local/man/man1
        install -m 755 -o 0 -g 0 -d $(ROOT)/usr/local/man/man1/
        install -m 755 -o 0 -g 0 cdp.1 $(ROOT)/usr/local/man/man1/cdp.1
```

In the example above, the commented lines were the original ones. The uncommented lines perform the same function, but also support installation in the root specified by the environment variable `ROOT`.

One point worth noting is that the `Makefile` now takes extra pains to make sure the proper directory structure exists before installing any files. This is often necessary, as build roots are deleted, in most cases, after the software has been packaged. This is why `install` is used with the `-d` option — to make sure the necessary directories have been created.

Let's see how it works:

```
# rpm -ba cdplayer-1.0.spec
* Package: cdplayer
Executing: %prep
+ cd /usr/src/redhat/BUILD
...
+ exit 0
```

```
Executing: %build
+ cd /usr/src/redhat/BUILD
+ cd cdplayer-1.0
...
+ exit 0
+ umask 022
Executing: %install
+ cd /usr/src/redhat/BUILD
+ cd cdplayer-1.0
+ make ROOT=/tmp/cdplayer install
install -m 755 -o 0 -g 0 -d /tmp/cdplayer/usr/local/bin/
install -m 755 -o 0 -g 0 cdp /tmp/cdplayer/usr/local/bin/cdp
ln -s ./cdp /tmp/cdplayer/usr/local/bin/cdplay
install -m 755 -o 0 -g 0 -d /tmp/cdplayer/usr/local/man/man1/
install -m 755 -o 0 -g 0 cdp.1 /tmp/cdplayer/usr/local/man/man1/cdp.1
+ exit 0
Executing: special doc
+ cd /usr/src/redhat/BUILD
+ cd cdplayer-1.0
+ DOCDIR=/tmp/cdplayer//usr/doc/cdplayer-1.0-1
+ rm -rf /tmp/cdplayer//usr/doc/cdplayer-1.0-1
+ mkdir -p /tmp/cdplayer//usr/doc/cdplayer-1.0-1
+ cp -ar README /tmp/cdplayer//usr/doc/cdplayer-1.0-1
+ exit 0
Binary Packaging: cdplayer-1.0-1
Finding dependencies...
Requires (2): libc.so.5 libncurses.so.2.0
usr/doc/cdplayer-1.0-1
usr/doc/cdplayer-1.0-1/README
usr/local/bin/cdp
usr/local/bin/cdplay
usr/local/man/man1/cdp.1
93 blocks
Generating signature: 0
Wrote: /usr/src/redhat/RPMS/i386/cdplayer-1.0-1.i386.rpm
+ umask 022
+ echo Executing: %clean
Executing: %clean
+ cd /usr/src/redhat/BUILD
+ cd cdplayer-1.0
+ exit 0
Source Packaging: cdplayer-1.0-1
cdplayer-1.0.spec
cdplayer-1.0.tgz
82 blocks
Generating signature: 0
```

```
Wrote: /usr/src/redhat/SRPMS/cdplayer-1.0-1.src.rpm
#
```

Looking over the output from the `%install` section, we first see that the `RPM_BUILD_ROOT` environment variable in the `make install` command, has been replaced with the path specified earlier in the spec file on the `BuildRoot:` line. The `ROOT` environment variable used in the makefile now has the appropriate value, as can be seen in the various `install` commands that follow.

Note, also, that we use `install`'s `-d` option to ensure that every directory in the path exists before we actually install the software. Unfortunately, we can't do this and install the file in one command.

Looking at the section labeled `Executing: special doc`, we find that RPM is doing something similar for us. It starts by making sure there is no pre-existing documentation directory. Next, RPM creates the documentation directory and copies files into it.

The remainder of this example is identical to that of a package being built without a build root being specified. However, although the output is identical, there is one crucial difference. When the binary package is created, instead of simply using each line in the `%files` list verbatim, RPM prepends the build root path first. If this wasn't done, RPM would attempt to find the files, relative to the system's root directory, and would, of course, fail. Because of the automatic prepending of the build root, it's important to `not` include the build root path in any `%files` list entry. Otherwise, the files would not be found by RPM, and the build would fail.

Although RPM has to go through a bit of extra effort to locate the files to be packaged, the resulting binary package is indistinguishable from the same package created without using a build root.

16.1.1 Some Things to Consider

Once the necessary modifications have been made to support a build root, it's necessary for the package builder to keep some issues in mind. The first is that the build root specified in the spec file can be overridden. RPM will set the build root (and therefore, the value of `$RPM_BUILD_ROOT`) to one of the following values:

- The value of `buildroot` in the spec file.
- The value of `buildroot` in an `rpmrc` file.
- The value following the `--buildroot` option on the command line.

Because of this, it's important that the spec file and the makefile be written in such a way that no assumptions about the build root are made. The main issue is that the build root must not be hard-coded anywhere. Always use the `RPM_BUILD_ROOT` environment variable!

Another issue to keep in mind is cleaning up after the build. Once software builds and is packaged successfully, it's probably no longer necessary to leave the build root in place. Therefore, it's a good idea to include the necessary commands in the spec file's `%clean` section. Here's an example:

```
%clean
rm -rf $RPM_BUILD_ROOT
```

Since RPM executes the `%clean` section after the binary package has been created, it's the perfect place to delete the build root tree. In the example above, that's exactly what we're doing. We're also doing the right thing by using the `RPM_BUILD_ROOT`, instead of a hard-coded path.

The last issue to keep in mind revolves around the `%clean` section we just created. At the start of the chapter, we mentioned that it's not a good idea to define a build root of "/". The `%clean` section is why: If the build root was set to "/", the `%clean` section would blow away your root filesystem! Keep in mind that this can bite you, even if the package's spec file doesn't specify "/" as a build root. It's possible to use the `--buildroot` option to specify a dangerous build root, too:

```
# rpm -ba --buildroot / cdplayer-1.0.spec
```

But for all the possible hazards using build roots can pose for the careless, it's the only way to prevent a build from disrupting the operation of certain packages on the build system. And for the person wanting to build packages without root access, it's the first of three steps necessary to accomplish the task. The next step is to direct RPM to build the software in a directory other than RPM's default one.

16.2 Having RPM Use a Different Build Area

While RPM's build root requires a certain amount of spec file and make file tweaking in order to get it working properly, directing RPM to perform the build in a different directory is a snap. The hardest part is to create the directories RPM will use during the build process.

16.2.1 Setting up a Build Area

RPM's build area consists of five directories in the top-level:

1. The `BUILD` directory is where the software is unpacked and built.
2. The `RPMS` directory is where the newly created binary package files are written.
3. The `SOURCES` directory contains the original sources, patches, and icon files.
4. The `SPECS` directory contains the spec files for each package to be built.
5. The `SRPMS` directory is where the newly created source package files are written.

The description of the `RPMS` directory above, is missing one key point. Since the binary package files are specific to an architecture, the directory actually contains one or more subdirectories, one for each architecture. It is in these subdirectories that RPM will write the binary package files.

Let's start by creating the directories. We can even do it with one command:

```
% pwd
/home/ed
% mkdir mybuild\
? mybuild/BUILD\
? mybuild/RPMS\
? mybuild/RPMS/i386\
? mybuild/SOURCES\
? mybuild/SPECS\
? mybuild/SRPMS
%
```

That's all there is to it. You may have noticed that we created a subdirectory to `RPMS` called `i386` — This is the architecture-specific subdirectory for Intel x86-based systems, which is our example build system.

The next step in getting RPM to use a different build area is telling RPM where the new build area is. And it's almost as easy as creating the build area itself.

16.2.2 Directing RPM to Use the New Build Area

All that's required to get RPM to start using the new build area is to define an alternate value for `topdir` in an rpmrc file. For the non-root user, this means putting the following line in a file called `.rpmrc`, located in your home directory:

```
topdir: <path>
```

By replacing "`<path>`" with the path to the new build area's top-level directory, RPM will attempt to use it the next time a build is performed. Using our newly created build area as an example, we'll set `topdir` to `/home/ed/mybuild`:

```
topdir: /home/ed/mybuild
```

That's all there is to it. Now it's time to try a build.

16.2.3 Performing a Build in a New Build Area

In the following example, a non-root user attempts to build the `cdplayer` package in a personal build area. If the user has modified `rpmrc` file entries to change the default build area, the command used to start the build is just like the one used by a root user. Otherwise, the `--buildroot` option will need to be used:

```
% cd /home/ed/mybuild/SPECS
% rpm -ba --buildroot /home/ed/mybuildroot cdplayer-1.0.spec
* Package: cdplayer
+ umask 022
Executing: %prep
+ cd /home/ed/mybuild/BUILD
+ cd /home/ed/mybuild/BUILD
+ rm -rf cdplayer-1.0
+ gzip -dc /home/ed/mybuild/SOURCES/cdplayer-1.0.tgz
+ tar -xvvf -
drwxrwxr-x root/users        0 Aug 20 20:58 1996 cdplayer-1.0/
-rw-r--r-- root/users    17982 Nov 10 01:10 1995 cdplayer-1.0/COPYING
...
+ cd /home/ed/mybuild/BUILD/cdplayer-1.0
+ chmod -R a+rX,g-w,o-w .
+ exit 0
Executing: %build
+ cd /home/ed/mybuild/BUILD
+ cd cdplayer-1.0
+ make
gcc -Wall -O2  -c -I/usr/include/ncurses  cdp.c
...
Executing: %install
+ cd /home/ed/mybuild/BUILD
+ make ROOT=/home/ed/mybuildroot/cdplayer install
install -m 755 -o 0 -g 0 -d /home/ed/mybuildroot/cdplayer/usr/local/bin/
install: /home/ed/mybuildroot/cdplayer: Operation not permitted
install: /home/ed/mybuildroot/cdplayer/usr: Operation not permitted
install: /home/ed/mybuildroot/cdplayer/usr/local: Operation not permitted
install: /home/ed/mybuildroot/cdplayer/usr/local/bin: Operation not
permitted
install: /home/ed/mybuildroot/cdplayer/usr/local/bin/: Operation not
permitted
make: *** [install] Error 1
Bad exit status
%
```

Things started off pretty well — The `%prep` section of the spec file unpacked the sources into the new build area, as did the `%build` section. The build was proceeding normally in the user-specified build area, and root access was not required. In the `%install` section, however, things started to fall apart. What happened?

Take a look at that `install` command. The two options, "`-o 0`" and "`-g 0`", dictate that the directories to be created in the build root are to be owned by the root account. Since the user performing this build did not have root access, the `install` failed, and rightly so.

OK, let's remove the offending options and see where that gets us. Here's the install section of the make file after our modifications:

```
install: cdp cdp.1.Z
install -m 755 -d $(ROOT)/usr/local/bin/
install -m 755 cdp $(ROOT)/usr/local/bin/cdp
rm -f $(ROOT)/usr/local/bin/cdplay
ln -s ./cdp $(ROOT)/usr/local/bin/cdplay
install -m 755 -d $(ROOT)/usr/local/man/man1/
install -m 755 cdp.1 $(ROOT)/usr/local/man/man1/cdp.1
```

We'll spare you from having to read through another build, but this time it completed successfully. Now, let's put our sysadmin hat on and install the newly built package:

```
# rpm -ivh cdplayer-1.0-1.i386.rpm
cdplayer        ##################################################
#
```

Well, that was easy enough. Let's take a look at some of the files and make sure everything looks OK. We know there are some files installed in `/usr/local/bin`, so let's check those:

```
# ls -al /usr/local/bin
-rwxr-xr-x 1 ed ed 40739 Sep 13 20:16 cdp*
lrwxrwxrwx 1 ed ed 47 Sep 13 20:34 cdplay -> ./cdp*
#
```

Looks pretty good... Wait a minute! What's up with the owner and group? The answer is simple: User `ed` ran the build, which executed the make file, which ran `install`, which created the files. Since `ed` created the files, they are owned by him.

This brings up an interesting point. Software must be installed with very specific file ownership and permissions. But a non-root user can't create files that are owned by anyone other than his or herself. What is a non-root user to do?

16.3 Specifying File Attributes

In cases where the package builder cannot create the files to be packaged with the proper ownership and permissions, the `%attr` macro can be used to make things right.

16.3.1 `%attr` — How Does It Work?

The `%attr` macro has the following format:

`%attr(<mode>, <user>, <group>)` *file*

- The `<mode>` is represented in traditional numeric fashion.
- The `<user>` is specified by the login name of the user. Numeric UIDs are *not* used, for reasons we'll explore in a moment.
- The `<group>` is specified by the group's name, as entered in `/etc/group`. Numeric GIDs are *not* used, either. Yes, we'll be discussing that, too!
- *<file>* represents the file. Shell-style globbing is supported.

There are a couple other wrinkles to using the `%attr` macro. If a particular file attribute doesn't need to be specified, that attribute can be replaced with a dash "-" and `%attr` will not change it. Say, for instance, that a package's files are installed with the permissions correctly set, as they almost always are. Instead of having to go to the trouble of entering the permissions for each and every file, each file can have the same `%attr`:

```
%attr(-, root, root)
```

This works for user and group specifications, as well.

The other wrinkle is that, although we've been showing the three file attributes separated by commas, in reality they could be separated by spaces as well. Whichever delimiter you choose, it pays to be consistent throughout a spec file.

Let's fix up `cdplayer` with a liberal sprinkling of `%attr`s. Here's what the `%files` list looks like after we've had our way with it:

```
%files
%attr(-, root, root) %doc README
%attr(4755, root, root) /usr/local/bin/cdp
%attr(-, root, root) /usr/local/bin/cdplay
%attr(-, root, rot) /usr/local/man/man1/cdp.1
```

A couple points are worth noting here. The line for `README` shows that multiple macros can be used on a line — in this case, one to set file attributes, and one to mark the file as being documentation. The `%attr` for `/usr/local/bin/cdp` declares the file to be setuid root. If it sends a shiver down your spine to know that anybody can create a package that will run setuid root when installed on your system — Good! Just because RPM makes it easy to install software doesn't mean that you should blindly install every package you find.

A single RPM command can quickly point out areas of potential problems and should be issued on any package file whose creators you don't trust:

```
% rpm -qlvp ../RPMS/i386/cdplayer-1.0-1.i386.rpm
drwxr-xr-x-  root  root   1024 Sep 13 20:16 /usr/doc/cdplayer-1.0-1
-rw-r--r---  root  root   1085 Nov 10 01:10 /usr/doc/cdplayer-1.0-1/README
-rwsr-xr-x-  root  root  40739 Sep 13 21:32 /usr/local/bin/cdp
lrwxrwxrwx-  root  root     47 Sep 13 21:32 /usr/local/bin/cdplay -> ./cdp
-rwxr-xr-x-  root   rot   4550 Sep 13 21:32 /usr/local/man/man1/cdp.1
%
```

Sure enough — there's that setuid root file. In this case we trust the package builder, so let's install it:

```
# rpm -ivh cdplayer-1.0-1.i386.rpm
cdplayer        ##################################################
group rot does not exist - using root
#
```

What's this about group "rot"? Looking back at the `rpm -qlvp` output, it looks like `/usr/local/man/man1/cdp.1` has a bogus group. Looking back even further, it's there in the `%attr` for that file. Must have been a typo. We could pretend that RPM used advanced artificial intelligence technology to come to the same conclusion as we did and made the appropriate change, but in reality, RPM simply used the only group identifier it could count on — root. RPM will do the same thing if it can't resolve a user specification.

Let's look at some of the files the package installed, including that worrisome setuid root file:

```
# ls /usr/local/bin
total 558
-rwsr-xr-x   1 root     root        40739 Sep 13 21:32 cdp*
lrwxrwxrwx   1 root     root           47 Sep 13 21:36 cdplay -> ./cdp*
#
```

RPM did just what it was supposed to — It gave the files the attributes specified by the `%attr` macros.

16.3.2 Betcha Thought We Forgot...

At the start of this section, we mentioned that the `%attr` macro wouldn't accept numeric uids or gids, and we promised to explain why. The reason is simply that, even if a package requires a certain user or group to own the package's files, the user may not have the same uid/gid from system to system. There — wasn't that simple?

In the next chapter, we'll discuss how to make your packaged software safe against modification by unscrupulous people. The name of the game is Pretty Good Privacy, and you'll see how signing packages with PGP is easier than you think!

Chapter 17

Adding PGP Signatures to a Package

In this chapter, we'll explore the steps required to add a digital signature to a package, using the software known as Pretty Good Privacy, or PGP. If you've used PGP before, you probably know everything you'll need to start signing packages in short order.

On the other hand, if you feel you need a bit more information on PGP before starting, please refer to Appendix G on page 427 for a brief introduction. Once you feel comfortable with PGP, come on back and learn how easy signing packages is...

17.1 Why Sign a Package?

The reason for signing a package is to provide authentication. With a signed package, it's possible for your user community to verify that the package they have was in your possession at some time and has not been changed since then. That "not changed" part is also a good reason to sign your packages, as digital signatures are a very robust way to guard against any modifications to the package.

Of course, as with anything else in life, adding a digital signature to a package isn't an ironclad guarantee that everything is right with the package, but it's about as sure a thing as humans can make it.

17.2 Getting Ready to Sign

OK, we've convinced you that signing packages is a good idea. Now we've got to make sure PGP and RPM are up to the task. As you might imagine, there are two parts to this process: one for PGP, and one for RPM. Let's get PGP ready first.

17.2.1 Preparing PGP: Creating a Key Pair

There is really very little to be done to PGP, assuming it's been installed properly. The only thing required is to generate a key pair. As mentioned in our mini-primer on PGP, the key pair consists of a secret key and a public key. In terms of signing packages, you will use your secret key to do the actual signing. Anyone interested in checking your signature will need your public key.

Creating a key pair is quite simple. All that's required is to issue a `pgp -kg` command, enter some information, and create some random bits. Here's an example key generating session:

```
# pgp -kg
Pretty Good Privacy(tm) 2.6.3a - Public-key encryption for the masses.
(c) 1990-96 Philip Zimmermann, Phil's Pretty Good Software. 1996-03-04
Uses the RSAREF(tm) Toolkit, which is copyright RSA Data Security, Inc.
Distributed by the Massachusetts Institute of Technology.
Export of this software may be restricted by the U.S. government.
Current time: 1996/10/31 00:42 GMT

Pick your RSA key size:
    1)   512 bits- Low commercial grade, fast but less secure
    2)   768 bits- High commercial grade, medium speed, good security
    3)  1024 bits- "Military" grade, slow, highest security


Choose 1, 2, or 3, or enter desired number of bits:  3
Generating an RSA key with a 1024-bit modulus.

You need a user ID for your public key.  The desired form for this
user ID is your name, followed by your E-mail address enclosed in
<angle brackets>, if you have an E-mail address.
For example:  John Q. Smith <12345.6789@compuserve.com>
Enter a user ID for your public key:

Example Key for RPM Book
You need a pass phrase to protect your RSA secret key.
Your pass phrase can be any sentence or phrase and may have many
words, spaces, punctuation, or any other printable characters.

Enter pass phrase:  <passphrase> (Not echoed)
Enter same pass phrase again:  <passphrase> (Still not echoed)

Note that key generation is a lengthy process.

We need to generate 952 random bits.  This is done by measuring the
time intervals between your keystrokes.  Please enter some random text
on your keyboard until you hear the beep:
```

(Many random characters were entered)

```
   0 * -Enough, thank you.
............................................
................................**** ...****
Pass phrase is good.  Just a moment....
Key signature certificate added.
Key generation completed.
#
```

Let's review each of the times PGP required information. The first thing PGP needed to know was the key size we wanted. Depending on your level of paranoia, simply choose an appropriate key size. In our example, we chose the "They're out to get me" key size of 1024 bits.

Next, we needed to choose a user ID for the key. The user ID should be descriptive and should also include sufficient information for someone to contact you. We entered `Example Key for RPM Book`, which goes against our suggestion, but is sufficient for the purposes of our example.

After entering a user ID, we needed to add a pass phrase. The pass phrase is used to protect your secret key, so it should be something difficult for someone else to guess. It should also be memorable for you, because if you forget your pass phrase, you won't be able to use your secret key! I entered a couple of words and numbers, put together in such a way that no one could ever guess I typed `rpm2kool4words`

Oops…

The pass phrase is entered twice, to ensure that no typing mistakes were made. PGP also performs some cursory checks on the pass phrase, ensuring that the phrase is at least somewhat secure.

Finally comes the strangest part of the key-generation process, creating random bits. This is done by measuring the time between keystrokes. The secret here is to *not* hold down a key so that it auto-repeats and to *not* wait several seconds between keystrokes. Simply start typing anything (even nonsense text) until PGP tells you you've typed enough.

After generating enough random bits, PGP takes a minute or so to create the key pair. Assuming everything completed successfully, you'll see an ending message similar to the one above. You'll also find, in a subdirectory of your login directory called `.pgp`, the following files:

```
# ls -al ~/.pgp
total 6
drwxr-xr-x   2 root     root         1024 Oct 30 19:44 .
drwxr-xr-x   5 root     root         1024 Oct 30 19:44 ..
-rw-------   1 root     root          176 Oct 30 19:44 pubring.bak
-rw-------   1 root     root          331 Oct 30 19:44 pubring.pgp
-rw-------   1 root     root          408 Oct 30 19:44 randseed.bin
-rw-------   1 root     root          509 Oct 30 19:44 secring.pgp
#
```

For those interested in learning exactly what each file is, feel free to consult any of the fine books on PGP. For the purposes of signing packages, all we need to know is where these files are located.

That's it! Now it's time to configure RPM to use your newly generated key.

17.2.2 Preparing RPM

RPM's configuration process is quite straightforward. It consists of adding a few `rpmrc` entries in a file of your choice. For more information on rpmrc files in general, please see Appendix B on page 363.

The entries that need to be added to an rpmrc file are:

- `signature`
- `pgp_name`
- `pgp_path`

Let's check out the entries.

`signature` The `signature` entry is used to select the type of signature that RPM is to use. At the time this book was written, the only legal value is `pgp`. So you would enter:

```
signature: pgp
```

`pgp_name` The `pgp_name` entry gives RPM the user ID of the key it is to sign packages with. In our key generation example, the user ID of the key we created was `Example Key for RPM Book`, so this is what our entry should look like:

```
pgp_name: Example Key for RPM Book
```

`pgp_path` The `pgp_path` entry is used to define the path to the directory where the keys are kept. This entry is not needed if the environment variable `PGPPATH` has been defined. In our example, we didn't move them from PGP's default location, which is in the subdirectory `.pgp`, off the user's login directory. Since we generated the key as `root`, our path is `/root/.pgp`. Therefore, our entry would look like this:

```
pgp_path: /root/.pgp
```

And that's it. Now it's time to sign some packages.

17.3 Signing Packages

There are three different ways to sign a package:

1. Signing a package at build-time.
2. Replacing the signature on an already-existing package.
3. Adding a signature to an already-existing package.

Lets take a look at each one, starting with build-time signing.

17.3.1 `--sign` – Sign a Package At Build-Time

The `--sign` option is used to sign a package as it is being built. When this option is added to an RPM build command, RPM will ask for your PGP pass phrase. If the pass phrase is correct, the build will proceed. If not, the build stops immediately.

Here's an example of `--sign` in action:

```
# rpm -ba --sign blather-7.9.spec
Enter pass phrase:  <passphrase> (Not echoed)

Pass phrase is good.
* Package: blather
...
Binary Packaging: blather-7.9-1
Finding dependencies...
...
Generating signature: 1002
Wrote: /usr/src/redhat/RPMS/i386/blather-7.9-1.i386.rpm
...
Source Packaging: blather-7.9-1
...
Generating signature: 1002
Wrote: /usr/src/redhat/SRPMS/blather-7.9-1.src.rpm
#
```

Once the pass phrase is entered, there's very little that is different from a normal build. The only obvious difference is the `Generating signature` message in both the binary and source packaging sections. The number following the message indicates that the signature added was created using PGP.[1]

Notice, that since RPM only signs the source and binary package files, only the `-bb`, and `-ba` options make any sense when used with `--sign`. This is due to the fact that only the `-bb` and `-ba` options create package files.

If we issue a quick signature check using RPM's `--checksig` option, we can see that there is, in fact, a PGP signature present:

[1] The list of possible signature types can be found in the RPM sources, specifically `signature.h` in RPM's `lib` subdirectory.

```
# rpm --checksig blather-7.9-1.i386.rpm
blather-7.9-1.i386.rpm:  size pgp md5 OK
#
```

It's clear to see that, in addition to the usual size and MD5 signatures, the package has a PGP signature.

Multiple Builds? No Problem!

You might be wondering how the `--sign` option would work if more than one package is to be built. Do you have to enter the pass phrase for every single package you build? The answer is no, as long as you build the packages with a single RPM command. Here's an example:

```
# rpm -ba --sign b*.spec
Enter pass phrase:  <passphrase> (Not echoed)

Pass phrase is good.
* Package: blather
...
Binary Packaging: blather-7.9-1
...
Generating signature: 1002
Wrote: /usr/src/redhat/RPMS/i386/blather-7.9-1.i386.rpm
...
Source Packaging: blather-7.9-1
...
Generating signature: 1002
Wrote: /usr/src/redhat/SRPMS/blather-7.9-1.src.rpm
...
* Package: bother
...
Binary Packaging: bother-3.5-1
...
Generating signature: 1002
Wrote: /usr/src/redhat/RPMS/i386/bother-3.5-1.i386.rpm
...
Source Packaging: bother-3.5-1
...
Generating signature: 1002
Wrote: /usr/src/redhat/SRPMS/bother-3.5-1.src.rpm
#
```

Using the `--sign` option makes it as easy to sign one package as it is to sign one hundred. But what happens if you need to change your public key? Will you need to rebuild every single one of your packages just to update the signature?

17.3.2 --resign – Replace a Package's Signature(s)

As we mentioned at the end of the previous section, from time to time it may be necessary to change your public key. Certainly this would be necessary if your key's security was compromised, but other, more mundane situations might require this.

Fortunately, RPM has an option that permits you to replace the signature on an already-built package, with a new one. The option is called `--resign`, and here's an example of its use:

```
# rpm --resign blather-7.9-1.i386.rpm
Enter pass phrase: <passphrase> (Not echoed)

Pass phrase is good.
blather-7.9-1.i386.rpm:
#
```

While the output is not as exciting as a package build, the `--resign` option can be a life-saver if you need to change a package's signature, and you don't want to rebuild.

As you might have guessed, the `--resign` option works properly on multiple package files:

```
# rpm --resign b*.rpm
Enter pass phrase: <passphrase> (Not echoed)

Pass phrase is good.
blather-7.9-1.i386.rpm:
bother-3.5-1.i386.rpm:
#
```

There Are Limits, However...

Unfortunately, older package files cannot be re-signed. The package file must be in version 3 format, at least. If you attempt to resign a package that is too old, here's what you'll see:

```
# rpm --resign blah.rpm
Enter pass phrase: <passphrase> (Not echoed)

Pass phrase is good.
blah.rpm:
blah.rpm: Can't re-sign v2.0 RPM
#
```

Not sure what version your package files are at? Just use the `file` command to check:

```
# file blather-7.9-1.i386.rpm
blather-7.9-1.i386.rpm:  RPM v3 bin i386 blather-7.9-1
#
```

The "v3" in `file`'s output indicates the package file format.

17.3.3 `--addsign` – Add a Signature To a Package

The `--addsign` option, as the name suggests, is used to add another signature to the package. It's pretty easy to see why someone would want to have a package that had been signed by the package builders. But what reason would there be for *adding* a signature to a package?

One reason to have more than one signature on a package would be to provide a means of documenting the path of ownership from the package builder to the end-user.

As an example, the division of a company creates a package and signs it with the division's key. The company's headquarters then checks the package's signature and adds the corporate signature to the package, in essence stating that the signed package received by them is authentic.

Continuing the example, the doubly-signed package makes its way to a retailer. The retailer checks the package's signatures and, when they check out, adds their signature to the package.

The package now makes its way to a company that wishes to deploy the package. After checking every signature on the package, they know that it is an authentic copy, unchanged since it was first created. Depending on the deploying company's internal controls, they may choose to add their own signature, thereby reassuring their employees that the package has received their corporate "blessing".

After this lengthy example, the actual output from the `--addsign` option is a bit anticlimactic:

```
# rpm --addsign blather-7.9-1.i386.rpm
Enter pass phrase:  <passphrase> (Not echoed)

Pass phrase is good.
blather-7.9-1.i386.rpm:
#
```

If we check the signatures of this package, we'll be able to see the multiple signatures:

```
# rpm --checksig blather-7.9-1.i386.rpm
```

```
blather-7.9-1.i386.rpm:  size pgp pgp md5 OK
#
```

The two pgp's in --checksig's output clearly shows that the package has been signed twice.

A Few Caveats

As with the --resign option, the --addsign option cannot do its magic on pre-V3 package files:

```
# rpm --addsign blah.rpm
Enter pass phrase:  <passphrase> (Not echoed)

Pass phrase is good.
blah.rpm:
blah.rpm:  Can't re-sign v2.0 RPM
#
```

OK, the error message may not be 100% accurate, but you get the idea.

Another thing to be aware of is that the --addsign option does not check for multiple identical signatures. Although it doesn't make much sense to do so, RPM will happily let you add the same signature as many times as you'd like:

```
# rpm --addsig blather-7.9-1.i386.rpm
Enter pass phrase:  <passphrase> (Not echoed)

Pass phrase is good.
blather-7.9-1.i386.rpm:
# rpm --addsig blather-7.9-1.i386.rpm
Enter pass phrase:  <passphrase> (Not echoed)

Pass phrase is good.
blather-7.9-1.i386.rpm:
# rpm --addsig blather-7.9-1.i386.rpm
Enter pass phrase:  <passphrase> (Not echoed)

Pass phrase is good.
blather-7.9-1.i386.rpm:
# rpm --addsig blather-7.9-1.i386.rpm
Enter pass phrase:  <passphrase> (Not echoed)

Pass phrase is good.
blather-7.9-1.i386.rpm:
# rpm --checksig blather-7.9-1.i386.rpm
```

```
blather-7.9-1.i386.rpm:  size pgp pgp pgp pgp md5 OK
#
```

As we can see from `--checksig`'s output, the package now has four identical signatures. Maybe this is the digital equivalent of pressing down extra hard while writing your name...

Chapter 18

Creating Subpackages

In this chapter, we will explore one of RPM's more interesting capabilities: the capability to create subpackages.

18.1 What Are Subpackages?

Very simply put, a subpackage is one of several package files created from a single spec file. RPM has the ability to create a main package, along with one or more subpackages. Subpackages may also be created without the main package. It's all up to the package builder.

18.2 Why Are They Needed?

If all the software in the world followed the usual "one source, one binary" structure, there would be no need for subpackages. After all, RPM handles the building and packaging of a program into a single package file just fine.

But software doesn't always conform to this simplistic structure. It's not unusual for software to support two or more different modes of operation. A client/server program, for example, comes in two flavors: a client, and a server.

And it can get more complicated than that. Sometimes software relies on another program so completely that the two cannot be built separately. The result is often several packages.

While it is certainly possible that some convoluted procedure could be devised to force these kinds of software into a single-package structure, it makes more sense to let RPM manage the creation of subpackages. Why? From the package builder's viewpoint, the main reason to use subpackages is to eliminate any duplication of effort.

By using subpackages, there's no need to maintain separate spec files and endure the resulting headaches when new versions of the software become available. By keeping everything in one spec file, new software versions can be quickly integrated, and every related subpackage rebuilt with a single command.

But that's enough of the preliminaries. Let's see how subpackages are created.

18.3 Our Example Spec File: Subpackages Galore!

Throughout this chapter, we'll be constructing a spec file that will consist of a number of subpackages. Let's start by listing the spec file's requirements:

- The main package name is to be `foo`.
- The version is to be 2.7.
- There are three subpackages:
  - The server subpackage, to be called `foo-server`.
  - The client subpackage, to be called `foo-client`.
  - The `baz` development library subpackage, to be called `bazlib`.
- The `bazlib` subpackage has a version of 5.6.
- Each subpackage will have its own `summary` and `description` tags.

Every spec file starts with a preamble, and this one is no different. In this case, the preamble will contain the following tags:

```
Name: foo
Version: 2.7
Release: 1
Source: foo-2.7.tgz
CopyRight: probably not
Summary: The foo app, and the baz library needed to build it
Group: bogus/junque
%description
This is the long description of the foo app, and the baz library needed to
build it...
```

As we can see, there's nothing different here: this is an ordinary spec file so far. Let's delve into things a bit more and see what we'll need to add to this spec file to create the subpackages we require.

18.4 Spec File Changes For Subpackages

The creation of subpackages is based strictly on the contents of the spec file. This doesn't mean that you'll have to learn an entirely new set of tags, conditionals, and directives in order to create subpackages. In fact, you'll only need to learn one.

The primary change to a spec file is structural and starts with the definition of a preamble for each subpackage.

18.4.1 The Subpackage's "Preamble"

When we introduced RPM package building in chapter 10, we said that every spec file contains a preamble. The preamble contains a variety of tags that define all sorts of information about the package. In a single package situation, the preamble must be at the start of the spec file. The spec file we're creating will have one there, too.

When creating a spec file that will build subpackages, each subpackage also needs a preamble of its own. These "sub-preambles" need only define information for the subpackage when that information differs from what is defined in the main preamble. For example, if we wanted to define an installation prefix for a subpackage, we would add the appropriate `prefix` tag to that subpackage's preamble. That subpackage would then be relocatable.

In a single-package spec file, there is nothing that explicitly identifies the preamble, other than its position at the top of the file. For subpackages, however, we need to be a bit more explicit. So we use the `%package` directive to identify the preamble for each subpackage.

The `%package` Directive

The `%package` directive actually performs two functions. As we mentioned above, it is used to denote the start of a subpackage's preamble. It also plays a role in forming the subpackage's name. As an example, let's say the main preamble contains the following `name` tag:

```
name: foo
```

Later in the spec file, there is a `%package` directive:

```
%package bar
```

This would result in the name of the subpackage being `foo-bar`.

In this way, it's easy to see the relationship of the subpackage to the main package (or other subpackages, for that matter). Of course, this naming convention might not be appropriate in every case. So there is an option to the `%package` directive for just this circumstance.

Adding `-n` To the `%package` directive

The `-n` option is used to change the final name of a subpackage from `mainpackage-subpackage` to `subpackage`. Let's modify the `%package` directive in our example above to be:

```
%package -n bar
```

The result is that the subpackage name would then be `bar` instead of `foo-bar`.

Updating Our Spec File

Let's apply some of our newly found knowledge to the spec file we're writing. Here's the list of subpackages that we need to create:

- The server subpackage, to be called `foo-server`.
- The client subpackage, to be called `foo-client`.
- The `baz` development library subpackage, to be called `bazlib`.

Since our package name is `foo`, and since the `%package` directive creates subpackage names by prepending the package name, the `%package` directives for the `foo-server` and `foo-client` subpackages would be written as:

```
%package server
```

and

```
%package client
```

Since the `baz` library's package name is *not* to start with `foo`, we need to use the `-n` option on its `%package` directive:

```
%package -n bazlib
```

Our requirements further state that `foo-server` and `foo-client` are to have the same version as the main package.

One of the time-saving aspects of using subpackages is that there is no need to duplicate information for each subpackage if it is already defined in the main package. Therefore, since the main package's preamble has a `version` tag defining the version as 2.7, the two subpackages that lack a `version` tag in their preambles will simply inherit the main package's version definition.

Since the `bazlib` subpackage's preamble contains a `version` tag, it must have its own unique version.

In addition, each subpackage must have its own `summary` tag.

So based on these requirements, our spec file now looks like this:

```
Name: foo
Version: 2.7
Release: 1
Source: foo-2.7.tgz
CopyRight: probably not
Summary: The foo app, and the baz library needed to build it
Group: bogus/junque
%description
```

```
This is the long description of the foo app, and the baz library needed to
build it...

%package server
Summary: The foo server

%package client
Summary: The foo client

%package -n bazlib
Version: 5.6
Summary: The baz library
```

We can see the subpackage structure starting to appear now.

Required Tags In Subpackages

There are a few more tags we should add to the subpackages in our example spec file. In fact, if these tags are *not* present, RPM will issue a most impressive warning:

```
# rpm -ba foo-2.7.spec
Package:  foo
Package:  foo-server
Field must be present :  Description
Field must be present :  Group
Package:  foo-client
Field must be present :  Description
Field must be present :  Group
Package:  bazlib
Field must be present :  Description
Field must be present :  Group

Spec file check failed!!
Tell rpm-list@redhat.com if this is incorrect.
#
```

Our spec file is incomplete. The bottom line is that each subpackage *must* have these three tags:

1. The `%description` tag.
2. The `group` tag.
3. The `summary` tag.

It's easy to see that the first two tags are required, but what about `summary`? Well, we lucked out on that one: we already included a `summary` for each subpackage in our example spec file.

Let's take a look at the `%description` tag first.

The `%description` Tag

As you've probably noticed, the `%description` tag differs from other tags. First of all, it starts with a percent sign. Secondly, its data can span multiple lines. The third difference is that the `%description` tag must include the name of the subpackage it describes. This is done by appending the subpackage name to the `%description` tag itself. So given these `%package` directives:

```
%package server
%package client
%package -n bazlib
```

our `%description` tags would start with:

```
%description server
%description client
%description -n bazlib
```

Notice that we've included the `-n` option in the `%description` tag for `bazlib`. This was intentional, as it makes the name completely unambiguous.

Our Spec File So Far...

OK, let's take a look at the spec file after we've added the appropriate `%descriptions`, along with `group` tags for each subpackage:

```
Name: foo
Version: 2.7
Release: 1
Source: foo-2.7.tgz
CopyRight: probably not
Summary: The foo app, and the baz library needed to build it
Group: bogus/junque
%description
This is the long description of the foo app, and the baz library needed to
build it...

%package server
Summary: The foo server
Group: bogus/junque
```

```
%description server
This is the long description for the foo server...

%package client
Summary: The foo client
Group: bogus/junque
%description client
This is the long description for the foo client...

%package -n bazlib
Version: 5.6
Summary: The baz library
Group: bogus/junque
%description -n bazlib
This is the long description for the bazlib...
```

Let's take a look at what we've done. We've created a main preamble as we normally would. We then created three additional preambles, each starting with a `%package` directive. Finally, we added a few tags to the subpackage preambles.

But what about `version` tags? Aren't the `server` and `client` subpackages missing them?

Not really. Remember that if a subpackage is missing a given tag, it will inherit the value of that tag from the main preamble. We're well on our way to having a complete spec file, but we aren't quite there yet.

Let's continue by looking at the next part of the spec file that changes when building subpackages.

18.4.2 The `%files` List

In an ordinary single-package spec file, the `%files` list is used to determine which files are actually going to be packaged. It is no different when building subpackages. What *is* different, is that there must be a `%files` list for each subpackage.

Since each `%files` list must be associated with a particular `%package` directive, we simply label each `%files` list with the name of the subpackage, as specified by each `%package` directive. Going back to our example, our `%package` lines were:

```
%package server
%package client
%package -n bazlib
```

Therefore, our `%files` lists should start with:

```
%files server
%files client
%files -n bazlib
```

In addition, we need the main package's `%files` list, which remains unnamed:

```
%files
```

The contents of each `%files` list is dictated entirely by the software's requirements. If, for example, a certain file needs to be packaged in more than one package, it's perfectly all right to include the filename in more than one list.

Controlling Packages With the `%files` List

The `%files` list wields considerable power over subpackages. It's even possible to prevent a package from being created by using the `%files` list. But is there a reason why you'd want to go to the trouble of setting up subpackages, only to keep one from being created?

Actually, there is. Take, for example, the case where client/server-based software is to be packaged. Certainly, it makes sense to create two subpackages: one for the client and one for the server. But what about the main package? Is there any need for it?

Quite often there's no need for a main package. In those cases, removing the main `%files` list entirely will result in no main package being built.

A Point Worth Noting

Please keep in mind that an empty `%files` list (ie, a `%files` list that contains no files) is *not* the same as not having a `%files` list at all. As we noted above, entirely removing a `%files` list results in RPM not creating that package. However, if RPM comes across a `%files` list with no files, it will happily create an empty package file.

This feature (which also works with subpackage `%files` lists) comes in handy when used in concert with conditionals. If a `%files` list is enclosed by a conditional, the package will be created (or not) based on the evaluation of the conditional.

Our Spec File So Far...

Ok, let's update our example spec file. Here's what it looks like after adding each of the subpackages' `%files` lists:

```
Name: foo
Version: 2.7
Release: 1
Source: foo-2.7.tgz
CopyRight: probably not
Summary: The foo app, and the baz library needed to build it
Group: bogus/junque
%description
This is the long description of the foo app, and the baz library needed to
build it...
```

```
%package server
Summary: The foo server
Group: bogus/junque

%package client
Summary: The foo client
Group: bogus/junque

%package -n bazlib
Version: 5.6
Summary: The baz library
Group: bogus/junque


%files
/usr/local/foo-file

%files server
/usr/local/server-file

%files client
/usr/local/client-file

%files -n bazlib
/usr/local/bazlib-file
```

As you can see we've added `%files` lists for:

- The main `foo` package.
- The `foo-server` subpackage.
- The `foo-client` subpackage.
- The `bazlib` subpackage.

Each package contains a single file.[1] If there was no need for a main package, we could simply remove the unnamed `%files` list. Keep in mind that even if you do not create a main package, the tags defined in the main package's preamble *will* appear somewhere – specifically, in the source package file.

Let's look at the last subpackage-specific part of the spec file: the install- and erase-time scripts.

[1] Hey, we said it was a simple example!

18.4.3 Install- and Erase-time Scripts

The install- and erase-time scripts, `%pre`, `%preun`, `%post`, and `%postun`, can all be named using exactly the same method as was used for the other subpackage-specific sections of the spec file. The script used during package verification, `%verifyscript`, can be made package-specific as well. Using the subpackage structure from our example spec file, we would end up with script definitions like:

- `%pre server`
- `%postun client`
- `%preun -n bazlib`
- `%verifyscript client`

Other than the change in naming, there's only one thing to be aware of when creating scripts for subpackages. It's important that you consider the possibility of scripts from various subpackages interacting with each other. Of course, this is simply good script-writing practice, even if the packages involved are not related.

Back At the Spec File...

Here we've added some scripts to our spec file. So that our example doesn't get too complex, we've just added preinstall scripts for each package:

```
Name: foo
Version: 2.7
Release: 1
Source: foo-2.7.tgz
CopyRight: probably not
Summary: The foo app, and the baz library needed to build it
Group: bogus/junque
%description
This is the long description of the foo app, and the baz library needed to
build it...

%package server
Summary: The foo server
Group: bogus/junque
%description server
This is the long description for the foo server...

%package client
Summary: The foo client
Group: bogus/junque
%description client
```

```
This is the long description for the foo client...

%package -n bazlib
Version: 5.6
Summary: The baz library
Group: bogus/junque
%description -n bazlib
This is the long description for the bazlib...


%pre
echo "This is the foo package preinstall script"

%pre server
echo "This is the foo-server subpackage preinstall script"

%pre client
echo "This is the foo-client subpackage preinstall script"

%pre -n bazlib
echo "This is the bazlib subpackage preinstall script"

%files
/usr/local/foo-file

%files server
/usr/local/server-file

%files client
/usr/local/client-file

%files -n bazlib
/usr/local/bazlib-file
```

As pre-install scripts go, these don't do very much. But they will allow us to see how subpackage-specific scripts can be defined.

Those of you that have built packages before probably realize that our spec file is missing something. Let's add that part now.

18.5 Build-Time Scripts: Unchanged For Subpackages

While creating subpackages changes the general structure of the spec file, there's one section that doesn't change: the build-time scripts. This means there is only one set of `%prep`, `%build`, and `%install` scripts in any spec file.

Of course, even if RPM doesn't require any changes to these scripts, you still might need to make some subpackage-related changes to them. Normally these changes are related to doing whatever is required to get the all the software unpacked, built, and installed. For example, if packaging client/server software, the software for both the client *and* the server must be unpacked, and then both the client *and* server binaries must be built and installed.

18.5.1 Our Spec File: One Last Look...

Let's add some build-time scripts and take a final look at the spec file:

```
Name: foo
Version: 2.7
Release: 1
Source: foo-2.7.tgz
CopyRight: probably not
Summary: The foo app, and the baz library needed to build it
Group: bogus/junque
%description
This is the long description of the foo app, and the baz library needed to
build it...

%package server
Summary: The foo server
Group: bogus/junque
%description server
This is the long description for the foo server...

%package client
Summary: The foo client
Group: bogus/junque
%description client
This is the long description for the foo client...

%package -n bazlib
Version: 5.6
Summary: The baz library
Group: bogus/junque
%description -n bazlib
This is the long description for the bazlib...

%prep
%setup

%build
make
```

```
%install
make install

%pre
echo "This is the foo package preinstall script"

%pre server
echo "This is the foo-server subpackage preinstall script"

#%pre client
#echo "This is the foo-client subpackage preinstall script"

%pre -n bazlib
echo "This is the bazlib subpackage preinstall script"

%files
/usr/local/foo-file

%files server
/usr/local/server-file

%files client
/usr/local/client-file

%files -n bazlib
/usr/local/bazlib-file
```

As you can see, the build-time scripts are about as simple as they can be.[2]

18.6 Building Subpackages

Now it's time to give our example spec file a try. The build process is not that much different from a single-package spec file:

```
# rpm -ba foo-2.7.spec
Package:  foo
Package:  foo-server
Package:  foo-client
Package:  bazlib
...
Executing:  %prep
...
Executing:  %build
```

[2]This is the advantage to making up an example. A more real-world spec file would undoubtedly have more interesting scripts.

```
...
Executing:  %install
...
Executing:  special doc
+ cd /usr/src/redhat/BUILD
+ cd foo-2.7
+ DOCDIR=//usr/doc/foo-2.7-1
+ DOCDIR=//usr/doc/foo-server-2.7-1
+ DOCDIR=//usr/doc/foo-client-2.7-1
+ DOCDIR=//usr/doc/bazlib-5.6-1
+ exit 0
Binary Packaging:  foo-2.7-1
Finding dependencies...
usr/local/foo-file
1 block
Generating signature:  0
Wrote:  /usr/src/redhat/RPMS/i386/foo-2.7-1.i386.rpm
Binary Packaging:  foo-server-2.7-1
Finding dependencies...
usr/local/server-file
1 block
Generating signature:  0
Wrote:  /usr/src/redhat/RPMS/i386/foo-server-2.7-1.i386.rpm
Binary Packaging:  foo-client-2.7-1
Finding dependencies...
usr/local/client-file
1 block
Generating signature:  0
Wrote:  /usr/src/redhat/RPMS/i386/foo-client-2.7-1.i386.rpm
Binary Packaging:  bazlib-5.6-1
Finding dependencies...
usr/local/bazlib-file
1 block
Generating signature:  0
Wrote:  /usr/src/redhat/RPMS/i386/bazlib-5.6-1.i386.rpm
...
Source Packaging:  foo-2.7-1
foo-2.7.spec
foo-2.7.tgz
4 blocks
Generating signature:  0
Wrote:  /usr/src/redhat/SRPMS/foo-2.7-1.src.rpm
#
```

Starting at the top, we start the build with the usual command. Immediately following the command, RPM indicates that four packages are to be built from this spec file. The `%prep`,

`%build`, and `%install` scripts then execute as usual.

Next, RPM executes its "special doc" internal script, even though we haven't declared any files to be documentation. It's worth noting, however, that the `DOCDIR` environment variables show that if the spec file *had* declared some of the files as documentation, RPM would have created the appropriate documentation directories for each of the packages.

At this point, RPM creates the binary packages. As we can see, each package contains the file defined in its `%files` list.

Finally, the source package file is created. It contains the spec file and the original sources, just like any other source package.

One spec file. One set of sources. One build command. Four packages.[3] All in all, a pretty good deal, isn't it?

18.6.1 Giving Subpackages the Once-Over

Let's take a look at our newly created packages. As with any other package, each subpackage should be tested by installing it on a system that has not had that software installed before. In this section, we'll just snoop around the subpackages and point out how they differ from packages built one to a spec file.

First, let's just look at each package's information:

```
# rpm -qip foo-2.7-1.i386.rpm
Name        : foo                    Distribution: (none)
Version     : 2.7                          Vendor: (none)
Release     : 1                        Build Date: Wed Nov 06 13:33:37 1996
Install date: (none)                   Build Host: foonly.rpm.org
Group       : bogus/junque             Source RPM: foo-2.7-1.src.rpm
Size        : 35
Summary     : The foo app, and the baz library needed to build it
Description :
This is the long description of the foo app, and the baz library needed to
build it...


# rpm -qip foo-server-2.7-1.i386.rpm
Name        : foo-server             Distribution: (none)
Version     : 2.7                          Vendor: (none)
Release     : 1                        Build Date: Wed Nov 06 13:33:37 1996
Install date: (none)                   Build Host: foonly.rpm.org
Group       : bogus/junque             Source RPM: foo-2.7-1.src.rpm
Size        : 42
Summary     : The foo server
Description :
This is the long description for the foo server...
```

[3] Five, if you count the source package.

```
# rpm -qip foo-client-2.7-1.i386.rpm
Name        : foo-client            Distribution: (none)
Version     : 2.7                         Vendor: (none)
Release     : 1                       Build Date: Wed Nov 06 13:33:37 1996
Install date: (none)                  Build Host: foonly.rpm.org
Group       : bogus/junque            Source RPM: foo-2.7-1.src.rpm
Size        : 42
Summary     : The foo client
Description :
This is the long description for the foo client...
```

```
# rpm -qip bazlib-5.6-1.i386.rpm
Name        : bazlib                Distribution: (none)
Version     : 5.6                         Vendor: (none)
Release     : 1                       Build Date: Wed Nov 06 13:33:37 1996
Install date: (none)                  Build Host: foonly.rpm.org
Group       : bogus/junque            Source RPM: foo-2.7-1.src.rpm
Size        : 38
Summary     : The baz library
Description :
This is the long description for the bazlib...
#
```

Here we've used RPM's query capability to display a list of summary information for each package. A few points are worth noting.

First, each package lists `foo-2.7-1.src.rpm` as its source package file. This is the only way to tell if two package files were created from the same set of sources. Trying to use a package's name as an indicator is futile, as the `bazlib` package shows us.

The next thing to notice is that the summaries and descriptions for each package are specific to that package. Since these tags were placed and named according to each package, that should be no surprise.

Finally, we can see that each package's version has been either "inherited" from the main package's preamble, or, as in the case of the `bazlib` package, the main package's version has been overridden by a `version` tag added to `bazlib`'s preamble.

If we look at the source package's information, we see that its information has been taken entirely from the main package's set of tags:

```
# rpm -qip foo-2.7-1.src.rpm
Name        : foo                   Distribution: (none)
Version     : 2.7                         Vendor: (none)
Release     : 1                       Build Date: Wed Nov 06 13:33:37 1996
Install date: (none)                  Build Host: foonly.rpm.org
Group       : bogus/junque            Source RPM: (none)
```

```
Size        : 1415
Summary     : The foo app, and the baz library needed to build it
Description :
This is the long description of the foo app, and the baz library needed to
build it...
#
```

It's easy to see that if there was no `%files` list for the main package, and therefore, no main package, the tags in the main preamble would still be used in the source package. This is why RPM enforces the requirement that the main preamble contain `copyright`, `%description`, and `group` tags. So, here's a word to the wise: Don't put something stupid in the main preamble's `%description` just to satisfy RPM. Your witty saying will be immortalized for all time in every source package you distribute.[4]

Verifying Subpackage-specific Install and Erase Scripts

The easiest way to verify that the `%pre` scripts we defined for each package were actually used is to simply install each package:

```
# rpm -Uvh foo-2.7-1.i386.rpm
foo                     This is the foo package preinstall script
##################################################


# rpm -Uvh foo-server-2.7-1.i386.rpm
foo-server              This is the foo-server subpackage preinstall script
##################################################


# rpm -Uvh foo-client-2.7-1.i386.rpm
foo-client              This is the foo-client subpackage preinstall script
##################################################


# rpm -Uvh bazlib-5.6-1.i386.rpm
bazlib                  This is the bazlib subpackage preinstall script
##################################################
#
```

As expected, the unique `%pre` script for each package has been included. Of course, if we hadn't wanted to actually install the packages, we could have used RPM's `--scripts` option to display the scripts:

```
# rpm -qp --scripts foo-2.7-1.i386.rpm
```

[4] Yes, the author found out about this hard way!

```
preinstall script:
echo "This is the foo-server subpackage preinstall script"

postinstall script:
(none)
preuninstall script:
(none)
postuninstall script:
(none)
verify script:
(none)
#
```

This approach might be a bit safer, particularly if installing the newly built package would disrupt operations on your build system.

Chapter 19

Building Packages for Multiple Architectures and Operating Systems

While RPM certainly makes packaging software as easy as possible, it doesn't end there. RPM gives you the tools you need to build a package on different types of computers. More importantly, RPM makes it possible to build packages on different types of computers *using a single spec file*. Those of you that have developed software for different computers know the importance of maintaining a single set of sources. RPM lets you continue that practice through the package building phase.

Before we get into RPM's capabilities, let's do a quick review of what is involved in developing software for different types of computer systems.

19.1 Architectures and Operating Systems: A Primer

From a software engineering standpoint, there are only two major differences between any two computer systems:

1. The architecture implemented by the computer's hardware.
2. The system software running on the computer.

The first difference is built into the computer. The architecture is the manner in which the computer system was designed. It includes the number and type of registers present in the processor, the number of machine instructions, what operations they perform, and so on. For example, every "PC" today, no matter who built it, is based on the Intel x86 architecture.

The second difference is more under our control. The operating system is software that controls how the system operates. Different operating systems have different methods of storing information on disk, different ways of implementing functions used by programs, and different hardware requirements.

As far as package building is concerned, two systems with the same architecture running two different operating systems, are as different as two systems with different architectures running the same operating system. In the first case, the software being packaged for different operating systems will differ due to the differences between the operating systems. In the second case, the software being packaged for different architectures will differ due to the underlying differences in hardware.[1]

RPM supports differences in architecture and operating system equally. If there is a tag, `rpmrc` file entry, or conditional that is used to support architectural differences, there is a corresponding tag, entry, or conditional that supports operating system differences.

19.1.1 Let's Just Call Them Platforms

In order to keep the duplication in this chapter to a minimum, we'll refer to a computer of a given architecture running a given operating system as a "platform". If another system differs in either aspect, it is considered a different platform.

OK, now that we've gotten through the preliminaries, let's look at RPM's multi-platform capabilities.

19.2 What Does RPM Do To Make Multi-Platform Packaging Easier?

As we mentioned above, RPM supports multi-platform package building through a set of tags, `rpmrc` file entries, and conditionals. None of these tools are difficult to use. In fact, the hardest part of multi-platform package building is figuring out how the software needs to be changed to support different platforms.

Let's take a look at each multi-platform tool RPM provides.

19.2.1 Automatic Detection of Build Platform

The first thing necessary for easy multi-platform package building is to identify which platform the package is to be built for. Except in the fairly esoteric case of cross-compilation, the build platform is the platform on which the package is built. RPM does this for you automatically, although it can be overridden at build-time.

[1] This is a somewhat simplistic view of the matter, as it's common for incompatibilities to crop up between two different implementations of the same operating system on different architectures.

19.2.2 Automatic Detection of Install Platform

The other important platform in package building is the platform on which the package is to be installed. Here again, RPM does this for you, though it's possible to override this when the package is installed.

But there is more to multi-platform package building than simply being able to determine the platform during package building and installation. The next component in multi-platform package building is a set of platform-dependent tags.

19.2.3 Platform-Dependent Tags

RPM uses a number of tags that control which platforms can build a package. These tags make it easier for the package builder to build multiple packages automatically, since the tags will keep RPM from attempting to build packages that are incompatible with the build platform.

19.2.4 Platform-Dependent Conditionals

While the platform-dependent tags provide a crude level of multi-platform control (i.e., the package will be built or not, depending on the tags and the build platform), RPM's platform-dependent conditionals provide a much finer level of control. By using these conditionals, it's possible to excise those parts of the spec file that are specific to another platform and replace them with one or more lines that are compatible with the build platform.

Now that we have a basic idea of RPM's multi-platform support features, let's take a more in-depth look at each one.

19.3 Build and Install Platform Detection

As we mentioned above, the first step to multi-platform package building is to identify the build platform. This is done by matching information from the build system's `uname` output against a number of `rpmrc` file entries.

Normally, it's not necessary to worry too much about the following `rpmrc` file entries, as RPM comes with a set of entries that support all platforms that currently run RPM. However, when adding support for new platforms, it will be necessary to use the following entries to add support for the new build platform.

19.3.1 Platform-Specific `rpmrc` Entries

Normally, the file `/usr/lib/rpmrc` contains the following `rpmrc` file entries. They can be overridden by entries in `/etc/rpmrc` or `~/.rpmrc`. This is discussed more completely in Appendix B.

Because each entry type is available in both architecture and operating system flavors, we'll just use *xxx* in place of `arch` and `os` in the following descriptions.

*xxx*_canon – Define Canonical Platform Name and Number

The *xxx*_canon entry is used to convert information obtained from the system running RPM into a canonical name and number that RPM will use internally. Here's the format:

```
xxx_canon:<label>:  <string> <value>
```

The <label> is compared against information from `uname(2)`. If a match is found, then <string> is used by RPM as the canonical name, and <value> is used as a unique numeric value. Here are two examples:

```
arch_canon:  sun4: sparc 3
os_canon: Linux: Linux 1
```

The `arch_canon` tag above is used to define the canonical architecture information for Sun Microsystems' SPARC architecture. In this case, the output from `uname` is compared against `sun4`. If there's a match, the canonical architecture name is set to `sparc` and the architecture number is set to `3`.

The `os_canon` tag above is used to define the canonical operating system information for the Linux operating system. In this case, the output from `uname` is compared against `Linux`. If there's a match, the canonical operating system name is set to `Linux`, and the operating system number is set to `1`.

The description above is not 100% complete — There is an additional step performed during the time RPM gets the system information from `uname`, and compares it against a canonical name. Next, let's look at the `rpmrc` file entry that comes into play during this intermediate step.

build*xxx*translate – Define Build Platform

The build*xxx*translate entry is used to define the build platform information. Specifically, these entries are used to create a table that maps information from `uname` to the appropriate architecture/operating system name.

The build*xxx*translate entry looks like this:

```
buildxxxtranslate:<label>:  <string>
```

The <label> is compared against information from `uname(2)`. If a match is found, then <string> is used by RPM as the build platform information, after it has been canonicalized by *xxx*_canon. Here are two examples:

```
buildarchtranslate: i586: i386
buildostranslate: Linux: Linux
```

The buildarchtranslate tag shown above is used to define the build architecture for an Intel Pentium (or i586 as it's shown here) processor. Any Pentium-based system will, by default, build packages for the Intel 80386 (or i386) architecture.

The buildostranslate tag shown above is used to define the build operating system for systems running the Linux operating system. In this case, the build operating system remains unchanged.

*xxx*_compat – Define Compatible Architectures

The *xxx*_compat entry is used to define which architectures and operating systems are compatible with one another. It is used at install-time only. The format of the entry is:

*xxx*_compat:<label>: <list>

The <label> is a name string as defined by an *xxx*_canon entry. The <list> following it consists of one or more names, also defined by arch_canon. If there is more than one name in the list, they should be separated by a space.

The names in the list are considered compatible to the name specified in the label.

```
arch_compat: i586: i486
arch_compat: i486: i386
os_compat: Linux: AIX
```

The arch_compat lines shown above illustrate how a family of upwardly compatible architectures may be represented. For example, if the build architecture was defined as an i586, the compatible architectures would be i486, and i386. However, if the build system was an i486, the only compatible architecture would be an i386.

While the os_compat line shown above is entirely fictional, its purpose would be to declare AIX compatible with Linux. If it were only that simple...

19.3.2 Overriding Platform Information At Build-Time

By using the rpmrc file entries discussed above, RPM usually makes the right decisions in selecting the build and install platforms. However, there might be times when RPM's selections aren't the best. Normally the circumstances are unusual, as in the case of cross-compiling software. In these cases, it is nice to have an easy way of overriding the build-time architecture and operating system.

The --buildarch and --buildos options can be used to set the build-time architecture and operating system rather than relying on RPM's automatic detection capabilities. These options are added to a normal RPM build command. One important point to remember is that, although RPM does try to find the specified architecture name, it does no checking as to the sanity of the entered architecture or operating system. For example, if you enter an entirely fictional operating system, RPM will issue a warning message, and then happily build a package for it.

Why? Wouldn't it make more sense for RPM to perform some sort of sanity check? In a word, no. One of RPM's main design goals was to never get in the way of the package builder. If someone has a need to override their build platform information, they should know what they're doing, and what the full implications of their actions are.

Bottom line: Unless you *know* why you need to use `--buildarch` or `--buildos`, you probably don't need to use them.

19.3.3 Overriding Platform Information At Install-Time

It's also possible to direct RPM to ignore platform information while a package is being installed. The `--ignorearch` and `--ignoreos` options, when added to any install or upgrade command, will direct RPM to proceed with the install or upgrade, even if the package's platform doesn't match the install platform.

Dangerous? Yes. But it can be indispensable in certain circumstances. Like the ability to override platform information at build-time, unless you know why you need to use `--ignorearch` or `--ignoreos`, you probably don't need to use them.

19.4 `optflags` – The Other `rpmrc` File Entry

While the `optflags` entry doesn't play a part in determining the build or install platform, it *does* play a role in multi-platform package building. The `optflags` entry is used to define a standard set of options that can be used during the build process, specifically during compilation.

The `optflags` entry looks like this:

```
optflags:<architecture> <value>
```

For example, assume the following `optflags` entries were placed in an `rpmrc` file:

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

If RPM was running on an Intel 80386-compatible architecture, the `optflags` value would be set to `-O2 -m486 -fno-strength-reduce`. If, however, RPM was running on a Sun SPARC-based system, `optflags` would be set to `-O2`.

This entry sets the `RPM_OPT_FLAGS` environment variable, which can be used in the `%prep`, `%build`, and `%install` scripts.

19.5 Platform-Dependent Tags

Once RPM has determined the build platform's information, that information can be used in the build process. The first way this information can be used is to determine whether a given package should be built on a given platform. This is done through the use of four tags that can be added to a spec file.

There can be many reasons to do this. For example, the software may not build correctly on a given platform. Or the software may be platform-specific, such that packaging the software on any other platform, while technologically possible, would really make no sense.

The real world is not always so clear-cut, so there might even be cases where a package should be built on, say, three different platforms, but no others. By carefully using the following tags, any conceivable situation can be covered.

Like the `rpmrc` file entries we've already discussed, there are identical tags for architecture and operating system, so we'll discuss them together.

19.5.1 The `exclude`*xxx* Tag

The `exclude`*xxx* tags are used to direct RPM to insure that the package does *not* attempt to build on the excluded platforms. One or more platforms may be specified after the `exclude`*xxx* tags, separated by either spaces or commas. Here are two examples:

```
ExcludeArch: sparc alpha
ExcludeOS: Irix
```

The first line prevents systems based on the Sun SPARC and Digital Alpha/AXP architectures from attempting to build the package. The second line insures that the package will not be built for the Silicon Graphics operating system, Irix.

If a build is attempted on an excluded architecture or operating system, the following message will be displayed, and the build will fail:

```
# rpm -ba cdplayer-1.0.spec
Arch mismatch!
cdplayer-1.0.spec doesn't build on this architecture
#
```

The `exclude`*xxx* tags are meant to explicitly prevent a finite set of architectures or operating systems from building a package. If your goal is to insure that a package will only build on *one* architecture, then you should use the `exclusive`*xxx* tags.

19.5.2 The `exclusive`*xxx* Tag

The `exclusive`*xxx* tags are used to direct RPM to only build the package on the specified platforms. These tags insure that, in the future, no brand-new platform will mistakenly attempt to build the package. RPM will build the package on the specified platforms only.

The syntax of the `exclusive`*xxx* tags is identical to `exclude`*xxx*:

```
ExclusiveArch: sparc alpha
ExclusiveOS: Irix
```

In the first line, the package will only build on a Sun SPARC or Digital Alpha/AXP system. In the second, the package will only be built on the Irix operating system.

The `exclusive`*xxx* tags are meant to explicitly allow a finite set of architectures or operating systems to build a package. If your goal is to insure that a package will *not* build on a specific platform, then you should use the `exclude`*xxx* tag.

19.6 Platform-Dependent Conditionals

Of course, the control exerted by the `exclude`*xxx* and `exclusive`*xxx* tags over package building is often too coarse. There may be packages, for example, that would build just fine on another platform, if only you could substitute a platform-specific patch file or change some paths in the `%files` list.

The key to exerting this kind of platform-specific control in the spec file is to use RPM's conditionals. The conditionals provide a general-purpose means of constructing a platform-specific version of the spec file during the actual build process.

19.6.1 Common Features of All Conditionals

There are a few things that are common to each conditional, so let's discuss them first. The first thing is that conditionals are block-structured. The second is that conditionals can be nested. Finally, conditionals can span any part of the spec file.

Conditionals Are Block Structured

Every conditional is block-structured – in other words, the conditional begins at a certain point within the spec file and continues some number of lines until it is ended. This forms a block that will be used or ignored, depending on the platform the conditional is checking for, as well as the build platform itself.

Every conditional starts with a line beginning with the characters `%if` and is followed by one of four platform-related conditions. Every conditional ends with a line containing the characters `%endif`.

Ignoring the platform-related conditions for a moment, here's an example of a conditional block:

```
%ifos Linux
Summary: This is a package for the Linux operating system
%endif
```

It's a one-line block, but a block nonetheless.

There's also another style of conditional block. As before, it starts with a `%if`, and ends with a `%endif`. But there's something new in the middle:

```
%ifos Linux
Summary: This is a package for the Linux operating system
%else
Summary: This is a package for some other operating system
%endif
```

Here we've replaced one `summary` tag with another.

Conditionals Can Be Nested

Conditionals can be nested — That is, the block formed by one conditional can enclose another conditional. Here's an example:

```
%ifarch i386

echo "This is an i386"

%ifos Linux
echo "This is a Linux system"
%else
echo "This is not a Linux system"
%endif

%else

echo "This is not an i386"

%endif
```

In this example, the first conditional block formed by the `%ifarch i386` line contains a complete `%ifos --- %else --- %endif` conditional. Therefore, if the build system was Intel-based, the `%ifos` conditional would be tested. If the build system was *not* Intel-based, the `%ifos` conditional would not be tested.

Conditionals Can Cross Spec File Sections

The next thing each conditional has in common is that there is no limit to the number of lines a conditional block can contain. You could enclose the entire spec file within a conditional, if you like. But it's much better to use conditionals to insert only the appropriate platform-specific contents.

Now that we have the basics out of the way, let's take a look at each of the conditionals and see how they work.

19.6.2 %if*xxx*

The `%if`*xxx* conditionals are used to control the inclusion of a block, as long as the platform-dependent information is true. Here are two examples:

`%ifarch i386 alpha`

In this case, the block following the conditional would be included only if the build architecture was `i386` or `alpha`.

`%ifos Linux`

This example would include the block following the conditional only if the operating system was `Linux`.

19.6.3 %ifn*xxx*

The `%ifn`*xxx* conditionals are used to control the inclusion of a block, as long as the platform-dependent information is *not* true. Here are two examples:

`%ifnarch i386 alpha`

In this case, the block following the conditional would be included only if the build architecture was *not* `i386` or `alpha`.

`%ifnos Linux`

This example would include the block following the conditional only if the operating system was *not* `Linux`.

19.7 Hints and Kinks

There isn't much in the way of hard and fast rules when it comes to multi-platform package building. But in general, the following uses of RPM's multi-platform capabilities seem to work the best:

- The `exclude`*xxx*`/exclusive`*xxx* tags are best used when it's known there's no reason for the package to be built on specific architectures.
- The `%if`*xxx* and `%ifn`*xxx* conditionals are most likely to be used in the following areas:
  - Controlling the inclusion of `%patch` macros for platform-specific patches.
  - Setting up platform-specific initialization prior to building the software.
  - Tailoring the `%files` list when the software creates platform-specific files.

Given that some software is more easily ported to different platforms than others, this list is far from complete. If there's one thing to remember about multi-platform package building, it's don't be afraid to experiment!

Chapter 20

Real-World Package Building

In chapter 11, we packaged a fairly simple application. Since our goal was to introduce package building, we kept things as simple as possible. However, things aren't always that simple in the real world.

In this chapter, we'll package a more complex application that will call on most of RPM's capabilities. We'll start with a general overview of the application and end with a completed package, just as you would if you were tasked with packaging an application that you'd not seen before.

So without further ado, let's meet amanda...

20.1 An Overview of Amanda

Amanda is a network backup utility. The name amanda stands for "Advanced Maryland Automatic Network Disk Archiver". If the word "Maryland" seems somewhat incongruous, it helps to realize that the program was developed at the University of Maryland by James Da Silva, and has subsequently been enhanced by many people around the world.

The sources are available at `ftp.cs.umd.edu`, in directory `/pub/amanda`. At the time of writing, the latest version of amanda is version 2.3.0. Therefore, it should come as no surprise that the amanda source `tar` file is called amanda-2.3.0.tar.gz.

As with most network-centric applications, amanda has a server component, and a client component. An amanda server controls how the various client systems are backed up to the server's tape drive. Each amanda client uses the operating system's native `dump` utility to perform the actual backup, which is then compressed and sent to the server. A server can back itself up simply by having the client software installed and configured, just like any other client system.

The software builds with `make`, and most customization is done in two `.h` files in the `config` subdirectory. A fair amount of documentation is available in the `doc` subdirectory. All in all, amanda is a typical non-trivial application.

Amanda can be built on several Unix-based operating systems. In this chapter, we'll build and package amanda for Red Hat Linux version 4.0.

20.2 Initial Building Without RPM

Since amanda can be built on numerous platforms, there needs to be some initial customization when first building the software. Since customization implies that mistakes will be made, we'll start off by building amanda without any involvement on the part of RPM.

But before we can build amanda, we have to get it and unpack it, first.

20.2.1 Setting Up A Test Build Area

As we mentioned above, the home FTP site for amanda is `ftp.cs.umd.edu`. The sources are in `/pub/amanda`.

After getting the sources, it's necessary to unpack them. We'll unpack them into RPM's `SOURCES` directory, so that we can keep all our work in one place:

```
# tar zxvf amanda-2.3.0.tar.gz
amanda-2.3.0/
amanda-2.3.0/COPYRIGHT
amanda-2.3.0/Makefile
amanda-2.3.0/README
...
amanda-2.3.0/man/amtape.8
amanda-2.3.0/tools/
amanda-2.3.0/tools/munge
#
```

As we saw, the sources unpacked into a directory called `amanda-2.3.0`. Let's rename that directory to `amanda-2.3.0-orig`, and unpack the sources again:

```
# ls
total 177
drwxr-xr-x 11 adm games 1024 May 19 1996 amanda-2.3.0/
-rw-r--r-- 1 root root 178646 Nov 20 10:42 amanda-2.3.0.tar.gz
# mv amanda-2.3.0 amanda-2.3.0-orig
# tar zxvf amanda-2.3.0.tar.gz
amanda-2.3.0/
amanda-2.3.0/COPYRIGHT
amanda-2.3.0/Makefile
amanda-2.3.0/README
...
amanda-2.3.0/man/amtape.8
```

```
amanda-2.3.0/tools/
amanda-2.3.0/tools/munge
# ls
total 178
drwxr-xr-x 11 adm games 1024 May 19 1996 amanda-2.3.0/
drwxr-xr-x 11 adm games 1024 May 19 1996 amanda-2.3.0-orig/
-rw-r--r-- 1 root root 178646 Nov 20 10:42 amanda-2.3.0.tar.gz
#
```

Now why did we do that? The reason lies in the fact that we will undoubtedly need to make changes to the original sources in order to get amanda to build on Linux. We'll do all our hacking in the `amanda-2.3.0` directory, and leave the `amanda-2.3.0-orig` untouched.

Since one of RPM's design features is to build packages from the original, unmodified sources, that means the changes we'll make will need to be kept as a set of patches. The `amanda-2.3.0-orig` directory will let us issue a simple recursive `diff` command to create our patches when the time comes.

Now that our sources are unpacked, it's time to work on building the software.

20.2.2 Getting Software to build

Looking at the `docs/INSTALL` file, we find that the steps required to get amanda configured and ready to build are actually fairly simple. The first step is to modify `tools/munge` to point to `cpp`, the C preprocessor.

Amanda uses CPP to create makefiles containing the appropriate configuration information. This approach is a bit unusual, but not unheard of. In `munge`, we find the following section:

```
# Customize CPP to point to your system's C preprocessor.

# if cpp is on your path:
CPP=cpp

# if cpp is not on your path, try one of these:
# CPP=/lib/cpp                  # traditional
# CPP=/usr/lib/cpp              # also traditional
# CPP=/usr/ccs/lib/cpp          # Solaris 2.x
```

Since `cpp` exists in `/lib` on Red Hat Linux, we need to change this part of `munge` to:

```
# Customize CPP to point to your system's C preprocessor.

# if cpp is on your path:
#CPP=cpp

# if cpp is not on your path, try one of these:
```

```
CPP=/lib/cpp                    # traditional
# CPP=/usr/lib/cpp              # also traditional
# CPP=/usr/ccs/lib/cpp          # Solaris 2.x
```

Next, we need to take a look in `config/` and create two files:

1. `config.h` — contains platform-specific configuration information
2. `options.h` — contains site-specific configuration information

There are a number of example `config.h` files for a number of different platforms. There is a Linux-specific version, so we copy that file to `config.h` and review it. After a few changes to reflect our Red Hat Linux environment, it's ready. Now let's turn our attention to `options.h`.

In the case of `options.h`, there's only one example file called `options.h-vanilla`. As the name implies, this is a basic file that contains a series of `#define`s that configure amanda for a typical environment. We'll need to make a few changes:

- Define the paths to common utility programs.
- Keep the programs from being named with the suffix `-2.3.0`.
- Define the directories where the programs should be installed.

While the first change is pretty much standard fare for anyone used to building software, the last two changes are really due to RPM. With RPM, there's no need to name the programs with a version-specific name, as RPM can easily upgrade to a new version and even downgrade back, if the new version doesn't work as well. The default paths amanda uses segregate the files so that they can be easily maintained. With RPM, there's no need to do this, since every file installed by RPM gets written into the database. In addition, Red Hat Linux systems adhere to the File System Standard, so any package destined for Red Hat systems should really be FSSTND-compliant, too. Fortunately for us, amanda was written to make these types of changes easy. But even if we had to hack an installation script, RPM would pick up the changes as part of its patch handling.

We'll spare you the usual discovery of typos, incompatibilities, and the resulting rebuilds. After an undisclosed number of iterations, our `config.h` and `options.h` files are perfect. Amanda builds:

```
# make
Making all in common-src
make[1]:  Entering directory '/usr/src/redhat/SOURCES/amanda-2.3.0/com
mon-src'
...
make[1]:  Leaving directory '/usr/src/redhat/SOURCES/amanda-2.3.0/man'
#
```

As we noted, amanda is constructed so that most of the time changes will only be necessary in `tools/munge`, and the two files in `config`. Our situation was no different — after all was said and done, that was all we needed to hack.

20.2.3 Installing and testing

As we all know, just because software builds doesn't mean that it's ready for prime-time. It's necessary to test it first. In order to test amanda, we need to install it. Amanda's makefile has an install target, so let's use that to get started. We'll also get a copy of the output, because we'll need that later:

```
# make install
Making install in common-src
...
make[1]: Entering directory '/usr/src/redhat/SOURCES/amanda-2.3.0/clien
t-src'
Installing Amanda client-side programs:
  install -c -o bin amandad /usr/lib/amanda
  install -c -o bin sendsize /usr/lib/amanda
  install -c -o bin calcsize /usr/lib/amanda
  install -c -o bin sendbackup-dump /usr/lib/amanda
  install -c -o bin sendbackup-gnutar /usr/lib/amanda
  install -c -o bin runtar /usr/lib/amanda
  install -c -o bin selfcheck /usr/lib/amanda
Setting permissions for setuid-root client programs:
  (cd /usr/lib/amanda ; chown root calcsize; chmod u+s calcsize)
  (cd /usr/lib/amanda ; chown root runtar; chmod u+s runtar)
...
Making install in server-src
Installing Amanda libexec programs:
  install -c -o bin taper /usr/lib/amanda
  install -c -o bin dumper /usr/lib/amanda
  install -c -o bin driver /usr/lib/amanda
  install -c -o bin planner /usr/lib/amanda
  install -c -o bin reporter /usr/lib/amanda
  install -c -o bin getconf /usr/lib/amanda
Setting permissions for setuid-root libexec programs:
  (cd /usr/lib/amanda ; chown root dumper; chmod u+s dumper)
  (cd /usr/lib/amanda ; chown root planner; chmod u+s planner)
Installing Amanda user programs:
  install -c -o bin amrestore /usr/sbin
  install -c -o bin amadmin /usr/sbin
  install -c -o bin amflush /usr/sbin
  install -c -o bin amlabel /usr/sbin
  install -c -o bin amcheck /usr/sbin
  install -c -o bin amdump /usr/sbin
```

```
  install -c -o bin amcleanup /usr/sbin
  install -c -o bin amtape /usr/sbin
Setting permissions for setuid-root user programs:
  (cd /usr/sbin ; chown root amcheck; chmod u+s amcheck)
...
Installing Amanda changer libexec programs:
  install -c -o bin chg-generic /usr/lib/amanda
...
Installing Amanda man pages:
  install -c -o bin amanda.8 /usr/man/man8
  install -c -o bin amadmin.8 /usr/man/man8
  install -c -o bin amcheck.8 /usr/man/man8
  install -c -o bin amcleanup.8 /usr/man/man8
  install -c -o bin amdump.8 /usr/man/man8
  install -c -o bin amflush.8 /usr/man/man8
  install -c -o bin amlabel.8 /usr/man/man8
  install -c -o bin amrestore.8 /usr/man/man8
  install -c -o bin amtape.8 /usr/man/man8
...
#
```

OK, no major problems there. Amanda does require a bit of additional effort to get everything running, though. Looking at `docs/INSTALL`, we follow the steps to get amanda running on our test system, as both a client and a server. As we perform each step, we note it for future reference:

- For the client:
  1. Set up a `~/.rhosts` file allowing the server to connect.
  2. Make the disk device files readable by the client.
  3. Make `/etc/dumpdates` readable and writeable by the client.
  4. Put an amanda entry in `/etc/services`.
  5. Put an amanda entry in `/etc/inetd.conf`.
  6. Issue a `kill -HUP` on inetd.
- For the server:
  1. Create a directory to hold the server configuration files.
  2. Modify the provided example configuration files to suit our site.
  3. Add crontab entries to run amanda nightly.
  4. Put an amanda entry in `/etc/services`.

Once everything is ready, we run a few tests. Everything performs flawlessly.[1] Looks like we've got a good build. Let's start getting RPM involved.

[1] Well, eventually it did!

20.3 Initial Building With RPM

Now that amanda has been configured, built, and is operational on our build system, it's time to have RPM take over each of these tasks. The first task is to have RPM make the necessary changes to the original sources. To do that, RPM needs a patch file.

20.3.1 Generating patches

The `amanda-2.3.0` directory tree is where we did all our work building amanda. We need to take all the work we've done in that directory tree and compare it against the original sources contained in the `amanda-2.3.0-orig` directory tree. But before we do that, we need to clean things up a bit.

Cleaning up the test build area

Looking through our work tree, it has all sorts of junk in it: emacs save files, object files, and the executable programs. In order to generate a clean set of patches, all these extraneous files must go. Looking over amanda's makefiles, there is a `clean` target that should take care of most of the junk:

```
# make clean
Making clean in common-src
...
rm -f *~ *.o *.a genversion version.c Makefile.out
...
Making clean in client-src
...
rm -f amandad sendsize calcsize sendbackup-dump
 sendbackup-gnutar runtar selfcheck  *~ *.o Makefile.out
...
Making clean in server-src
...
rm -f amrestore amadmin amflush amlabel amcheck amdump
 amcleanup amtape taper dumper driver planner reporter
 getconf *~ *.o Makefile.out
...
Making clean in changer-src
...
rm -f chg-generic *~ *.o Makefile.out
...
Making clean in man
...
rm -f *~ Makefile.out
...
#
```

Looking in the `tools` and `config` directories where we did all our work, we see there are still emacs save files there. A bit of studying confirms that the makefiles don't bother to clean these two directories. That's a nice touch because a `make clean` won't wipe out old copies of the config files, giving you a chance to go back to them in case you've botched something. However, in our case, we're sure we won't need the save files, so out they go:

```
# cd /usr/src/redhat/SOURCES/amanda-2.3.0
# find .  -name "* " -exec rm -vf  \;
./config/config.h
./config/options.h
./tools/munge
#
```

We let `find` take a look at the whole directory tree, just in case there was something still out there that we'd forgotten about. As you can see, the only save files are from the three files we've been working on.

You'll note that we've left our modified `munge` file, as well as the `config.h` and `options.h` files we so carefully crafted. That's intentional, as we want to make sure those changes are applied when RPM patches the sources. Everything looks pretty clean, so it's time to make the patches.

Actually Generating patches

This step is actually pretty anticlimactic:

```
# diff -uNr amanda-2.3.0-orig/ amanda-2.3.0/ > amanda-2.3.0-linux.patch
#
```

With that one command, we've compared each file in the untouched directory tree (`amanda-2.3.0-orig`) with the directory tree we've been working in (`amanda-2.3.0`). If we've done our homework, the only things in the patch file should be related to the files we've changed. Let's take a look through it to make sure:

```
# cd /usr/src/redhat/SOURCES
# cat amanda-2.3.0-linux.patch
diff -uNr amanda-2.3.0-orig/config/config.h amanda-2.3.0/config/config.h
--- amanda-2.3.0-orig/config/config.h Wed Dec 31 19:00:00 1969
+++ amanda-2.3.0/config/config.h Sat Nov 16 16:22:47 1996
@@ -0,0 +1,52 @@
...
diff -uNr amanda-2.3.0-orig/config/options.h amanda-2.3.0/config/options.h
--- amanda-2.3.0-orig/config/options.h Wed Dec 31 19:00:00 1969
+++ amanda-2.3.0/config/options.h Sat Nov 16 17:08:57 1996
@@ -0,0 +1,211 @@
```

```
...
diff -uNr amanda-2.3.0-orig/tools/munge amanda-2.3.0/tools/munge
--- amanda-2.3.0-orig/tools/munge Sun May 19 22:11:25 1996
+++ amanda-2.3.0/tools/munge Sat Nov 16 16:23:50 1996
@@ -35,10 +35,10 @@
 # Customize CPP to point to your system's C preprocessor.

 # if cpp is on your path:
-CPP=cpp
+# CPP=cpp

 # if cpp is not on your path, try one of these:
-# CPP=/lib/cpp                # traditional
+CPP=/lib/cpp                 # traditional
 # CPP=/usr/lib/cpp            # also traditional
 # CPP=/usr/ccs/lib/cpp        # Solaris 2.x
#
```

The patch file contains complete copies of our `config.h` and `options.h` files, followed by the changes we've made to `munge`. Looks good! Time to hand this grunt work over to RPM.

20.3.2 Making a first-cut spec file

Since amanda comes in two parts, it's obvious we'll need to use subpackages: one for the client software, and one for the server. Given that, and the fact that the first part of any spec file consists of tags that are easily filled in, let's sit down and fill in the blanks, tag-wise:

```
Summary: Amanda Network Backup System
Name: amanda
Version: 2.3.08
Release: 1
Group: System/Backup
Copyright: BSD-like, but see COPYRIGHT file for details
Packager: Edward C. Bailey <bailey@rpm.org>
URL: http://www.cs.umd.edu/projects/amanda/
Source: ftp://ftp.cs.umd.edu/pub/amanda/amanda-2.3.0.tar.gz
Patch: amanda-2.3.0-linux.patch
%description
Amanda is a client/server backup system.  It uses standard tape
devices and networking, so all you need is any working tape drive
and a network.  You can use it for local backups as well.
```

That part was pretty easy. We set the package's release number to 1. We'll undoubtedly be changing that as we continue work on the spec file. You'll notice that we've included a

URL tag line; the Uniform Resource Locator there points to the homepage for the amanda project, making it easier for the user to get additional information on amanda.

The Source tag above includes the name of the original source tar file and is preceded by the URL pointing to the file's primary location. Again, this makes it easy for the user to grab a copy of the sources from the software's "birthplace".

Finally, the patch file that we've just created gets a line of its own on the Patch tag line. Next, let's take a look at the tags for our two subpackages. Let's start with the client:

```
%package client
Summary: Client-side Amanda package
Group: System/Backup
Requires: dump
%description client
The Amanda Network Backup system contains software necessary to
automatically perform backups across a network.  Amanda consists of
two packages -- a client (this package), and a server:

The client package enable a network-capable system to have its
filesystems backed up by a system running the Amanda server.

NOTE: In order for a system to perform backups of itself, install both
the client and server packages!
```

The %package directive names the package. Since we wanted the subpackages to be named amanda-<something>, we didn't use the -n option. This means our client subpackage will be called amanda-client, just as we wanted. RPM requires unique summary, %description, and group tags for each subpackage, so we've included them. Of course, it would be a good idea even if RPM *didn't* require them — we've used the tags to provide client-specific information.

The requires tag is the only other tag in the client subpackage. Since amanda uses dump on the client system, we included this tag so that RPM will ensure that the dump package is present on client systems.

Next, let's take a look at the tags for the server subpackage:

```
%package server
Summary: Server-side Amanda package
Group: System/Backup
%description server
The Amanda Network Backup system contains software necessary to
automatically perform backups across a network.  Amanda consists of
two package -- a client, and a server (this package):

The server package enables a network-capable system to control one
or more Amanda client systems performing backups.  The server system
will direct all backups to a locally attached tape drive.  Therefore,
```

```
the server system requires a tape drive.

NOTE: In order for a system to perform backups of itself, install both
the client and server packages!
```

No surprises here, really. You'll note that the server subpackage has no `requires` tag for the dump package. The reason for that is due to a design decision we've made. Since amanda is comprised of a client and a server component, in order for the server system to perform backups of itself, the client component must be installed. Since we've already made the client subpackage require `dump`, we've already covered the bases.

Since an amanda server cannot back itself up without the client software, why don't we have the server subpackage require the client subpackage? Well, that could be done, but the fact of the matter is that there are cases where an amanda server won't need to back itself up. So the server subpackage needs no package requirements.

Adding the build-time scripts

Next we need to add the build-time scripts. There's really not much to them:

```
%prep
%setup

%build
make

%install
make install
```

The `%prep` script consists of one line containing the simplest flavor of `%setup` macro. Since we only need `%setup` to unpack one set of sources, there are no options we need to add.

The `%build` script is just as simple, with the single `make` command required to build amanda.

Finally, the `%install` script maintains our singe-line trend for build-time scripts. Here a simple `make install` will put all the files where they need to be for RPM to package them.

Adding `%files` Lists

The last part of our initial attempt at a spec file is a `%files` list for each package the spec file will build. Since we're planning on a client and a server subpackage, we'll need two `%files` lists. For the time being, we'll just add the `%files` lines — we'll be adding the actual filenames later:

```
%files client

%file server
```

There's certainly more to come, but this is enough to get us started. And the first thing we want RPM to do is to unpack the amanda sources.

20.3.3 Getting the original sources unpacked

In keeping with a step-by-step approach, RPM has an option that let's us stop the build process after the `%prep` script has run. Let's give the `-bp` option a try, and see how things look:

```
# rpm -bp amanda-2.3.0.spec
* Package: amanda
* Package: amanda-client
* Package: amanda-server
+ umask 022
+ echo Executing: %prep
Executing: %prep
+ cd /usr/src/redhat/BUILD
+ cd /usr/src/redhat/BUILD
+ rm -rf amanda-2.3.0
+ gzip -dc /usr/src/redhat/SOURCES/amanda-2.3.0.tar.gz
+ tar -xvvf -
drwxr-xr-x 3/20              0 May 19 22:10 1996 amanda-2.3.0/
-rw-r--r-- 3/20           1389 May 19 22:11 1996 amanda-2.3.0/COPYRIGHT
-rw-r--r-- 3/20           1958 May 19 22:11 1996 amanda-2.3.0/Makefile
-rw-r--r-- 3/20          11036 May 19 22:11 1996 amanda-2.3.0/README
...
-rw-r--r-- 3/20           2010 May 19 22:11 1996 amanda-2.3.0/man/amtape.8
drwxr-xr-x 3/20              0 May 19 22:11 1996 amanda-2.3.0/tools/
-rwxr-xr-x 3/20           2437 May 19 22:11 1996 amanda-2.3.0/tools/munge
+ [ 0 -ne 0 ]
+ cd amanda-2.3.0
+ cd /usr/src/redhat/BUILD/amanda-2.3.0
+ chown -R root.root .
+ chmod -R a+rX,g-w,o-w .
+ exit 0
#
```

By looking at the output, it would be pretty hard to miss the fact that the sources were unpacked. If we look in RPM's default build area (`/usr/src/redhat/BUILD`), we'll see an amanda directory tree:

```
# cd /usr/src/redhat/BUILD/
# ls -l
total 3
drwxr-xr-x 11 root root 1024 May 19 1996 amanda-2.3.0
```

```
#
```

After a quick look around, it seems like the sources were unpacked properly. But wait — where are our carefully crafted configuration files in `config`? Why isn't `tools/munge` modified?

20.3.4 Getting patches properly applied

Ah, perhaps our `%prep` script was a bit *too* simple. We need to apply our patch. So let's add two things to our spec file:

1. A `patch` tag line pointing to our patch file
2. A `%patch` macro in our `%prep` script

Easy enough. At the top of the spec file, along with the other tags, let's add:

```
Patch: amanda-2.3.0-linux.patch
```

Then we'll make our `%prep` script look like this:

```
%prep
%setup
%patch -p 1
```

There, that should do it. Let's give that `-bp` option another try:

```
# rpm -bp amanda-2.3.0.spec
* Package: amanda
* Package: amanda-client
* Package: amanda-server
+ umask 022
+ echo Executing: %prep
Executing: %prep
+ cd /usr/src/redhat/BUILD
+ cd /usr/src/redhat/BUILD
+ rm -rf amanda-2.3.0
+ gzip -dc /usr/src/redhat/SOURCES/amanda-2.3.0.tar.gz
+ tar -xvvf -
drwxr-xr-x 3/20            0 May 19 22:10 1996 amanda-2.3.0/
-rw-r--r-- 3/20         1389 May 19 22:11 1996 amanda-2.3.0/COPYRIGHT
-rw-r--r-- 3/20         1958 May 19 22:11 1996 amanda-2.3.0/Makefile
-rw-r--r-- 3/20        11036 May 19 22:11 1996 amanda-2.3.0/README
...
-rw-r--r-- 3/20         2010 May 19 22:11 1996 amanda-2.3.0/man/amtape.8
```

```
drwxr-xr-x 3/20             0 May 19 22:11 1996 amanda-2.3.0/tools/
-rwxr-xr-x 3/20          2437 May 19 22:11 1996 amanda-2.3.0/tools/munge
+ [ 0 -ne 0 ]
+ cd amanda-2.3.0
+ cd /usr/src/redhat/BUILD/amanda-2.3.0
+ chown -R root.root .
+ chmod -R a+rX,g-w,o-w .
+ echo Patch #0:
Patch #0:
+ patch -p1 -s
+ exit 0
#
```

Not much difference, until the very end, where we see the patch being applied. Let's take a look into the build area and see if our configuration files are there:

```
# cd /usr/src/redhat/BUILD/amanda-2.3.0/config
# ls -l
total 58
-rw-r--r--   1 root     root         7518 May 19  1996 config-common.h
-rw-r--r--   1 root     root         1846 Nov 20 20:46 config.h
-rw-r--r--   1 root     root         2081 May 19  1996 config.h-aix
-rw-r--r--   1 root     root         1690 May 19  1996 config.h-bsdi1
...
-rw-r--r--   1 root     root         1830 May 19  1996 config.h-ultrix4
-rw-r--r--   1 root     root            0 Nov 20 20:46 config.h.orig
-rw-r--r--   1 root     root         7196 Nov 20 20:46 options.h
-rw-r--r--   1 root     root         7236 May 19  1996 options.h-vanilla
-rw-r--r--   1 root     root            0 Nov 20 20:46 options.h.orig
#
```

Much better. Those zero-length `.orig` files are a dead giveaway that patch has been here, as are the dates on `config.h`, and `options.h`. In the `tools` directory, `munge` has been modified, too. These sources are ready for building!

20.3.5 Letting RPM do the Building

We know that the sources are ready. We know that the `%build` script is ready. There shouldn't be much in the way of surprises if we let RPM build amanda. Let's use the `-bc` option to stop things after the `%build` script completes:

```
# rpm -bc amanda-2.3.0.spec
* Package: amanda
* Package: amanda-client
* Package: amanda-server
...
```

```
 echo Executing: %build
Executing: %build
+ cd /usr/src/redhat/BUILD
+ cd amanda-2.3.0
+ make
Making all in common-src
make[1]: Entering directory ‘/usr/src/redhat/BUILD/amanda-2.3.0/common-src’
../tools/munge Makefile.in Makefile.out
make[2]: Entering directory ‘/usr/src/redhat/BUILD/amanda-2.3.0/common-src’
cc  -g  -I. -I../config   -c error.c -o error.o
cc  -g  -I. -I../config   -c alloc.c -o alloc.o
...
Making all in man
make[1]: Entering directory ‘/usr/src/redhat/BUILD/amanda-2.3.0/man’
../tools/munge Makefile.in Makefile.out
make[2]: Entering directory ‘/usr/src/redhat/BUILD/amanda-2.3.0/man’
make[2]: Nothing to be done for ‘all’.
make[2]: Leaving directory ‘/usr/src/redhat/BUILD/amanda-2.3.0/man’
make[1]: Leaving directory ‘/usr/src/redhat/BUILD/amanda-2.3.0/man’
+ exit 0
#
```

As we thought, no surprises. A quick look through the build area shows a full assortment of binaries, all ready to be installed. So it seems that the most natural thing to do next would be to let RPM install amanda.

20.3.6 Letting RPM do the Installing

And that's just what we're going to do! Our `%install` script has the necessary `make install` command, so let's give it a shot:

```
# rpm -bi amanda-2.3.0.spec
* Package: amanda
* Package: amanda-client
* Package: amanda-server
...
 echo Executing: %build
Executing: %build
+ cd /usr/src/redhat/BUILD
+ cd amanda-2.3.0
+ make
Making all in common-src
make[1]: Entering directory ‘/usr/src/redhat/BUILD/amanda-2.3.0/common-src’
../tools/munge Makefile.in Makefile.out
make[2]: Entering directory ‘/usr/src/redhat/BUILD/amanda-2.3.0/common-src’
cc  -g  -I. -I../config   -c error.c -o error.o
```

```
cc  -g  -I. -I../config   -c alloc.c -o alloc.o
...
+ umask 022
+ echo Executing: %install
Executing: %install
+ cd /usr/src/redhat/BUILD
+ cd amanda-2.3.0
+ make install
Making install in common-src
make[1]: Entering directory `/usr/src/redhat/BUILD/amanda-2.3.0/common-src'
...
  install -c -o bin amrestore.8 /usr/man/man8
  install -c -o bin amtape.8 /usr/man/man8
make[2]: Leaving directory `/usr/src/redhat/BUILD/amanda-2.3.0/man'
make[1]: Leaving directory `/usr/src/redhat/BUILD/amanda-2.3.0/man'
+ exit 0
+ umask 022
+ echo Executing: special doc
Executing: special doc
+ cd /usr/src/redhat/BUILD
+ cd amanda-2.3.0
+ DOCDIR=//usr/doc/amanda-2.3.0-1
+ DOCDIR=//usr/doc/amanda-client-2.3.0-1
+ DOCDIR=//usr/doc/amanda-server-2.3.0-1
+ exit 0
#
```

Everything looks pretty good. At this point, the amanda software, built by RPM, has been installed on the build system. Since performed all the configuration steps before, when we were manually building amanda, everything should still be configured properly to test this new build.[2] So why don't we give the new binaries a try?

20.3.7 Testing RPM's Handiwork

After a quick double-check to ensure that all the configuration steps were still in place from our manual build, we reran our tests. No problems were found. It's time to build some packages!

20.4 Package Building

OK, let's go for broke and tell RPM to do the works, including the creation of the binary and source packages:

[2] Of course, if the process of installing the software changed some necessary config files, they would have to be redone, but in this case it didn't happen.

```
# rpm -ba amanda-2.3.0.spec
* Package: amanda
* Package: amanda-client
* Package: amanda-server
...
 echo Executing: %build
Executing: %build
+ cd /usr/src/redhat/BUILD
+ cd amanda-2.3.0
+ make
Making all in common-src
...
+ echo Executing: %install
Executing: %install
+ cd /usr/src/redhat/BUILD
+ cd amanda-2.3.0
+ make install
Making install in common-src
...
+ echo Executing: special doc
Executing: special doc
...
Binary Packaging: amanda-client-2.3.0-1
Finding dependencies...
Requires (1): dump
1 block
Generating signature: 0
Wrote: /usr/src/redhat/RPMS/i386/amanda-client-2.3.0-1.i386.rpm
Binary Packaging: amanda-server-2.3.0-1
Finding dependencies...
1 block
Generating signature: 0
Wrote: /usr/src/redhat/RPMS/i386/amanda-server-2.3.0-1.i386.rpm
+ umask 022
+ echo Executing: %clean
Executing: %clean
+ cd /usr/src/redhat/BUILD
+ cd amanda-2.3.0
+ exit 0
Source Packaging: amanda-2.3.0-1
amanda-2.3.0.spec
amanda-2.3.0-linux.patch
amanda-2.3.0.tar.gz
374 blocks
Generating signature: 0
Wrote: /usr/src/redhat/SRPMS/amanda-2.3.0-1.src.rpm
```

```
#
```

Great! Let's take a look at our handiwork:

```
# cd /usr/src/redhat/RPMS/i386/
# ls -l
total 2
-rw-r--r-- 1 root  root   1246 Nov 20 21:19 amanda-client-2.3.0-1.i386.rpm
-rw-r--r-- 1 root  root   1308 Nov 20 21:19 amanda-server-2.3.0-1.i386.rpm
#
rpm-first-look
```

Hmmm, those binary packages look sort of small. We'd better see what's in there:

```
# rpm -qilp amanda-*-1.i386.rpm
Name        : amanda-client        Distribution: (none)
Version     : 2.3.0                      Vendor: (none)
Release     : 1                      Build Date: Wed Nov 20 21:19:44 1996
Install date: (none)                 Build Host: moocow.rpm.org
Group       : System/Backup          Source RPM: amanda-2.3.0-1.src.rpm
Size        : 0
Summary     : Client-side Amanda package
Description :
The Amanda Network Backup system contains software necessary to
automatically perform backups across a network.  Amanda consists of
two packages -- a client (this package), and a server:

The client package enable a network-capable system to have its
filesystems backed up by a system running the Amanda server.

NOTE: In order for a system to perform backups of itself, install both
the client and server packages!
(contains no files)

Name        : amanda-server        Distribution: (none)
Version     : 2.3.0                      Vendor: (none)
Release     : 1                      Build Date: Wed Nov 20 21:19:44 1996
Install date: (none)                 Build Host: moocow.rpm.org
Group       : System/Backup          Source RPM: amanda-2.3.0-1.src.rpm
Size        : 0
Summary     : Server-side Amanda package
Description :
The Amanda Network Backup system contains software necessary to
automatically perform backups across a network.  Amanda consists of
two package -- a client, and a server (this package):
```

```
The server package enables a network-capable system to control one
or more Amanda client systems performing backups.  The server system
will direct all backups to a locally attached tape drive.  Therefore,
the server system requires a tape drive.

NOTE: In order for a system to perform backups of itself, install both
the client and server packages!
(contains no files)

#
```

What do they mean, `(contains no files)`? The spec file has perfectly good `%files` lists...

Oops.

20.4.1 Creating the `%files` list

Everything was going so smoothly, we forgot that the `%files` lists were going to need files. No problem, we just need to put the filenames in there, and we'll be all set. But is it *really* that easy?

How to find the installed files?

Luckily, it's not too bad. Since we saved the output from our first `make install`, we can see the filenames as they're installed. Of course, it's important to make sure the install output is valid. Fortunately for us, amanda didn't require much fiddling by the time we got it built and tested. If it had, we would have had to get more recent output from the installation phase.

It's time for more decisions. We have one list of installed files, and two `%files` lists. It would be silly to put all the files in both `%files` lists, so we have to decide which file goes where.

This is where experience with the software really pays off, because the wrong decision made here can result in awkward, ill-featured packages. Here's the `%files` list we came up with for the client subpackage:

```
%files client
/usr/lib/amanda/amandad
/usr/lib/amanda/sendsize
/usr/lib/amanda/calcsize
/usr/lib/amanda/sendbackup-dump
/usr/lib/amanda/selfcheck
/usr/lib/amanda/sendbackup-gnutar
/usr/lib/amanda/runtar
README
```

```
COPYRIGHT
docs/INSTALL
docs/SYSTEM.NOTES
docs/WHATS.NEW
```

The files in /usr/lib/amanda are all the client-side amanda programs, so that part was easy. The remaining files are part of the original source archive. Amanda doesn't install them, but they contain information that users should see.

Realizing that RPM can't package these files specified as they are, let's leave the client %files list for a moment, and check out the list for the server subpackage:

```
%files server
/usr/sbin/amadmin
/usr/sbin/amcheck
/usr/sbin/amcleanup
/usr/sbin/amdump
/usr/sbin/amflush
/usr/sbin/amlabel
/usr/sbin/amrestore
/usr/sbin/amtape
/usr/lib/amanda/taper
/usr/lib/amanda/dumper
/usr/lib/amanda/driver
/usr/lib/amanda/planner
/usr/lib/amanda/reporter
/usr/lib/amanda/getconf
/usr/lib/amanda/chg-generic
/usr/man/man8/amanda.8
/usr/man/man8/amadmin.8
/usr/man/man8/amcheck.8
/usr/man/man8/amcleanup.8
/usr/man/man8/amdump.8
/usr/man/man8/amflush.8
/usr/man/man8/amlabel.8
/usr/man/man8/amrestore.8
/usr/man/man8/amtape.8
README
COPYRIGHT
docs/INSTALL
docs/KERBEROS
docs/SUNOS4.BUG
docs/SYSTEM.NOTES
docs/TAPE.CHANGERS
docs/WHATS.NEW
docs/MULTITAPE
example
```

The files in /usr/sbin are programs that will be run by the amanda administrator in order to perform backups and restores. The files in /usr/lib/amanda are the server-side programs that do the actual work during backups. Following that are a number of man pages: one for each program to be run by the amanda administrator, and one with an overview of amanda.

Bringing up the rear are a number of files that are not installed, but would be handy for the amanda administrator to have available. There is some overlap with the files that will be part of the client subpackage, but the additional files here discuss features that would interest only amanda administrators. Included here is the example subdirectory, which contains a few example configuration files for the amanda server.

As in the client %files list, these last files can't be packaged by RPM as we've listed them. We need to use a few more of RPM's tricks to get them packaged.

Applying Directives

Since we'd like the client subpackage to include those files that are not normally installed, and since the files are documentation, let's use the %doc directive on them. That will accomplish two things:

1. When the client subpackage is installed, it will direct RPM to place them in a package-specific directory in /usr/doc
2. It will tag the files as being documentation, making it possible for users to easily track down the documentation with a simple rpm -qd command

In the course of looking over the %files lists, it becomes apparent that the directory /usr/lib/amanda will contain only files from the two amanda subpackages. If the subpackages are erased, the directory will remain, which won't hurt anything, but it isn't as neat as it could be. But if we add the directory to the list, RPM will automatically package every file in the directory. Since the files in that directory are part of both the client and the server subpackages, we'll need to use the %dir directive to instruct RPM to package only the directory.

After these changes, here's what the client %files list looks like now:

```
%files client
%dir /usr/lib/amanda/
/usr/lib/amanda/amandad
/usr/lib/amanda/sendsize
/usr/lib/amanda/calcsize
/usr/lib/amanda/sendbackup-dump
/usr/lib/amanda/selfcheck
/usr/lib/amanda/sendbackup-gnutar
/usr/lib/amanda/runtar
%doc README
%doc COPYRIGHT
%doc docs/INSTALL
```

```
%doc docs/SYSTEM.NOTES
%doc docs/WHATS.NEW
```

We've also applied the same directives to the server `%files` list:

```
%files server
/usr/sbin/amadmin
/usr/sbin/amcheck
/usr/sbin/amcleanup
/usr/sbin/amdump
/usr/sbin/amflush
/usr/sbin/amlabel
/usr/sbin/amrestore
/usr/sbin/amtape
%dir /usr/lib/amanda/
/usr/lib/amanda/taper
/usr/lib/amanda/dumper
/usr/lib/amanda/driver
/usr/lib/amanda/planner
/usr/lib/amanda/reporter
/usr/lib/amanda/getconf
/usr/lib/amanda/chg-generic
/usr/man/man8/amanda.8
/usr/man/man8/amadmin.8
/usr/man/man8/amcheck.8
/usr/man/man8/amcleanup.8
/usr/man/man8/amdump.8
/usr/man/man8/amflush.8
/usr/man/man8/amlabel.8
/usr/man/man8/amrestore.8
/usr/man/man8/amtape.8
%doc README
%doc COPYRIGHT
%doc docs/INSTALL
%doc docs/KERBEROS
%doc docs/SUNOS4.BUG
%doc docs/SYSTEM.NOTES
%doc docs/TAPE.CHANGERS
%doc docs/WHATS.NEW
%doc docs/MULTITAPE
%doc example
```

You'll note that we neglected to use the `%doc` directive on the man page files. The reason is that RPM automatically tags any file destined for `/usr/man` as documentation. Now our spec file has a complete set of tags, the two subpackages are defined, it has build-time scripts that work, and now, `%files` lists for each subpackage. Why don't we try that build again?

```
# rpm -ba amanda-2.3.0.spec
* Package: amanda
* Package: amanda-client
* Package: amanda-server
...
 echo Executing: %build
Executing: %build
+ cd /usr/src/redhat/BUILD
+ cd amanda-2.3.0
+ make
Making all in common-src
...
+ echo Executing: %install
Executing: %install
+ cd /usr/src/redhat/BUILD
+ cd amanda-2.3.0
+ make install
Making install in common-src
...
+ echo Executing: special doc
Executing: special doc
...
Binary Packaging: amanda-client-2.3.0-6
Finding dependencies...
Requires (3): libc.so.5 libdb.so.2 dump
usr/doc/amanda-client-2.3.0-6
usr/doc/amanda-client-2.3.0-6/COPYRIGHT
usr/doc/amanda-client-2.3.0-6/INSTALL
...
usr/lib/amanda/sendbackup-gnutar
usr/lib/amanda/sendsize
1453 blocks
Generating signature: 0
Wrote: /usr/src/redhat/RPMS/i386/amanda-client-2.3.0-6.i386.rpm
Binary Packaging: amanda-server-2.3.0-6
Finding dependencies...
Requires (2): libc.so.5 libdb.so.2
usr/doc/amanda-server-2.3.0-6
usr/doc/amanda-server-2.3.0-6/COPYRIGHT
usr/doc/amanda-server-2.3.0-6/INSTALL
...
usr/sbin/amrestore
usr/sbin/amtape
3404 blocks
Generating signature: 0
Wrote: /usr/src/redhat/RPMS/i386/amanda-server-2.3.0-6.i386.rpm
```

```
...
Source Packaging: amanda-2.3.0-6
amanda-2.3.0.spec
amanda-2.3.0-linux.patch
amanda-rpm-instructions.tar.gz
amanda-2.3.0.tar.gz
393 blocks
Generating signature: 0
Wrote: /usr/src/redhat/SRPMS/amanda-2.3.0-6.src.rpm
#
```

If we take a quick look at the client and server subpackages, we find that, sure enough, this time they contain files:

```
# cd /usr/src/redhat/RPMS/i386/
# ls -l amanda-*
-rw-r--r-- 1 root  root  211409 Nov 21 15:56 amanda-client-2.3.0-1.i386.rpm
-rw-r--r-- 1 root  root  512814 Nov 21 15:57 amanda-server-2.3.0-1.i386.rpm
# rpm -qilp amanda-*
Name        : amanda-client         Distribution: (none)
Version     : 2.3.0                       Vendor: (none)
Release     : 1                       Build Date: Thu Nov 21 15:55:59 1996
Install date: (none)                  Build Host: moocow.rpm.org
Group       : System/Backup           Source RPM: amanda-2.3.0-1.src.rpm
Size        : 737101
Summary     : Client-side Amanda package
Description :
The Amanda Network Backup system contains software necessary to
automatically perform backups across a network.  Amanda consists of
two packages -- a client (this package), and a server:

The client package enable a network-capable system to have its
filesystems backed up by a system running the Amanda server.

NOTE: In order for a system to perform backups of itself, install both
the client and server packages!

/usr/doc/amanda-client-2.3.0-1
/usr/doc/amanda-client-2.3.0-1/COPYRIGHT
/usr/doc/amanda-client-2.3.0-1/INSTALL
...
/usr/lib/amanda/sendbackup-gnutar
/usr/lib/amanda/sendsize

Name        : amanda-server         Distribution: (none)
Version     : 2.3.0                       Vendor: (none)
Release     : 1                       Build Date: Thu Nov 21 15:55:59 1996
```

```
Install date: (none)                    Build Host: moocow.rpm.org
Group       : System/Backup             Source RPM: amanda-2.3.0-1.src.rpm
Size        : 1733825
Summary     : Server-side Amanda package
Description :
The Amanda Network Backup system contains software necessary to
automatically perform backups across a network.  Amanda consists of
two package -- a client, and a server (this package):

The server package enables a network-capable system to control one
or more Amanda client systems performing backups.  The server system
will direct all backups to a locally attached tape drive.  Therefore,
the server system requires a tape drive.

NOTE: In order for a system to perform backups of itself, install both
the client and server packages!

/usr/doc/amanda-server-2.3.0-1
/usr/doc/amanda-server-2.3.0-1/COPYRIGHT
/usr/doc/amanda-server-2.3.0-1/INSTALL
...
/usr/sbin/amrestore
/usr/sbin/amtape
#
```

We're finally ready to test these packages!

20.4.2 Testing those first packages

The system we've built the packages on already has amanda installed. This is due to the build process itself. However, we can install the new packages on top of the already-existing files:

```
# cd /usr/src/redhat/RPMS/i386
# rpm -ivh amanda-*-1.i386.rpm
amanda-client       ##################################################
amanda-server       ##################################################
#
```

Running some tests, it looks like everything is running well. But back in chapter 11, specifically section 11.5.2 on page 137, we mentioned that it was possible to install a newly-built package on the build system, and not realize that the package was missing files. Well, there's another reason why installing the package on the build-system for testing is a bad idea. Let's bring our packages to a different system, test them there, and see what happens.

Installing the Package On A Different System

Looks like we're almost through. Let's install the packages on another system that had not previously run amanda, and test it there:

```
# rpm -ivh amanda-*-1.i386.rpm
amanda-client        ##################################################
amanda-server        ##################################################
#
```

The install went smoothly enough. However, testing did not. Why? Nothing was set up! The server configuration files, the `inetd.conf` entry for the client, everything was missing. If we stop and thing about it for a moment that makes sense: we had gone through all those steps on the build system, but none of those steps can be packaged as files.

After following the steps in the installation instructions, everything works. While we could expect users to do most of the grunt work associated with getting amanda configured, RPM *does* have the ability to run scripts when packages are installed and erased. Why don't we use that feature to make life easier for our users?

20.4.3 Finishing Touches

At this point in the build process, we're on the home stretch. The software builds correctly and is packaged. It's time to stop looking at things from a "build the software" perspective, and time to starting looking at things from a "package the software" point of view.

The difference lies in looking at the packages from the user's perspective. Does the package install easily, or does it require a lot of effort to make it operative? When the package is removed, does it clean up after itself, or does it leave bits and pieces strewn throughout the filesystem?

Let's put a bit more effort into this spec file, and make life easier on our users.

Creating Install Scripts

When it comes to needing post-installation configuration, amanda certainly is no slouch! We'll work on the client first. Let's look at a section of the script we wrote, comment on it, and move on:

```
%post client

# See if they've installed amanda before...
# If they have, none of this should be necessary...

if [ "$1" = 1 ];
then
```

First, we start the script with a `%post` statement, and indicate that this script is for the `client` subpackage. As the comments indicate, we only want to perform the following tasks if this is the first time the client subpackage has been installed. To do this, we use the first and only argument passed to the script. It is a number indicating how many instances of this package will be installed after the current installation is complete.

If the argument is equal to 1, that means that no other instances of the client subpackage are presently installed, and that this one is the first. Let's continue:

```
# Set disk devices so that bin can read them
# (This is actually done on Red Hat Linux; only need to add bin to
#  group disk)

if grep "^disk::.*bin" /etc/group > /dev/null
then
        true
else

# If there are any members in group disk, add bin after a comma...
 sed -e 's/\(^disk::[0-9]\{1,\}:.\{1,\}\)/\1,bin/' /etc/group > /etc/grou
p.tmp

# If there are no members in group disk, add bin...
sed -e 's/\(^disk::[0-9]\{1,\}:$\)/\1bin/' /etc/group.tmp > /etc/group

# clean up!
 rm -f /etc/group.tmp
fi
```

One of amanda's requirements is that the user ID running the dumps on the client needs to be able to read from every disk's device file. The folks at Red Hat have done half the work for us by creating a group `disk` and giving that group read/write access to every disk device. Since our dumpuser is `bin`, we only need to add `bin` to the `disk` group. Two lines of `sed`, and we're done!

The next section is related to the last. It also focuses on making sure `bin` can access everything it needs while doing backups:

```
# Also set /etc/dumpdates to be writable by group disk

chgrp disk /etc/dumpdates
chmod g+w /etc/dumpdates
```

Since amanda uses `dump` to obtain the backups, and since `dump` keeps track of the backups in `/etc/dumpdates`, it's only natural that `bin` will need read/write access to the file. In a perfect world, `/etc/dumpdates` would have already been set to allow group `disk` to read and write, but we had to do it ourselves. It's not a big problem, though.

Next, we need to create the appropriate network-related entries, so that amanda clients can communicate with amanda servers, and vice versa:

```
# Add amanda line to /etc/services

if grep "^amanda" /etc/services >/dev/null
then
        true
else
        echo "amanda    10080/udp # Added by package amanda-client" >>
/etc/services
fi
```

By using `grep` to look for lines that begin with the letters `amanda`, we can easily see if `/etc/services` is already configured properly. It it isn't, we simply append a line to the end.

We also added a comment so that sysadmins will know where the entry came from, and either take our word for it or issue an `rpm -q --scripts amanda-client` command and see for themselves. We did it all on one line because it makes the script simpler.

Let's look at the rest of the network-related part of this script:

```
# Add amanda line to /etc/inetd.conf

if grep "^amanda" /etc/inetd.conf >/dev/null
then
        true
else
        echo "amanda dgram udp wait bin /usr/lib/amanda/amandad amandad
  # added by package amanda-client" >>/etc/inetd.conf

# Kick inetd

if [ -f /var/run/inetd.pid ];
then
        kill -HUP `cat /var/run/inetd.pid`
fi
fi
fi
```

Here, we've used the same approach to add an entry to `/etc/inetd.conf`. We then hup `inetd` so the change will take affect, and we're done!

Oh, and that last `fi` at the end? That's to close the `if [ "$1" = 1 ]` at the start of the script. Now let's look at the server's post-install script:

```
%post server
```

```
# See if they've installed amanda before...

if [ "$1" = 1 ];
then

# Add amanda line to /etc/services

if grep "^amanda" /etc/services >/dev/null
then
        true
else
        echo "amanda    10080/udp # Added by package amanda-server"
 >>/etc/services
fi

fi
```

That was short! And this huge difference brings up a good point about writing install scripts: It's important to understand what you as the package builder should do for the user, and what they should do for themselves.

In the case of the client package, every one of the steps performed by the post-install script was something that a fairly knowledgeable user could have done. But each of these steps have one thing in common. No matter how the user configures amanda, these steps will never change. And given the nature of client/server applications, there's a good chance that *many* more amanda client packages will be installed than amanda servers. Would *you* like to be tasked with installing this package on twenty systems, and performing each of the steps we've automated, twenty times? We thought not.

There is one step that we did *not* automate for the client package. The step we left out is the creation of a `.rhosts` file. Since this file must contain the name of the amanda server, we have no way of knowing what the file should look like. Therefore, that's one step we can't automate.

The server's post-install script is so short because there's little else that can be automated. The other steps required to set up an amanda server include:

1. Choosing a configuration name, which requires user input
2. Creating a directory to hold the server configuration files, named according to the configuration name, which depends on the first step
3. Modifying example configuration files to suit the site, which requires user input
4. Adding crontab entries to run amanda nightly, which requires user input

Since every step depends on the user making decisions, the best way to handle them is to not handle them at all. Let the user do it!

Creating Uninstall Scripts

Where there are install scripts, there are uninstall scripts. While there is no ironclad rule to that effect, it is a good practice. Following this practice, we have an uninstall script for the client package, and one for the server. Let's take the client first:

```
%postun client

# First, see if we're the last amanda-client package on the system...
# If not, then we don't need to do this stuff...

if [ "$1" = 0 ];
then
```

As before, we start out with a declaration of the type of script this is, and which subpackage it is for. Following that we have an `if` statement similar to the one we used in the install scripts, save one difference. Here, we're comparing the argument against zero. The reason is that we are trying to see if there will be zero instances of this package after the uninstall is complete. If this is the case, the remainder of the script needs to be run, since there are no other amanda client packages left.

Next, we remove `bin` from the `disk` group:

```
# First, get rid of bin from the disk group...

if grep "^disk::.*bin" /etc/group > /dev/null
then

#       Nuke bin at the end of the line...
        sed -e 's/\(^disk::[0-9]\{1,\}:.\{1,\}\),bin$/\1/' /etc/group
 > /etc/group.tmp

#       Nuke bin on the line by itself...
        sed -e 's/\(^disk::[0-9]\{1,\}:\)bin$/\1/' /etc/group.tmp
 > /etc/group1.tmp

#       Nuke bin in the middle of the line...
        sed -e 's/\(^disk::[0-9]\{1,\}:.\{1,\}\),bin,\(.\{1,\}\)/\1,\2/'
 /etc/group1.tmp > /etc/group2.tmp

#       Nuke bin at the start of the line...
        sed -e 's/\(^disk::[0-9]\{1,\}:\)bin,\(.\{1,\}\)/\1\2/'
 /etc/group2.tmp > /etc/group

#       Clean up after ourselves...
        rm -f /etc/group.tmp /etc/group1.tmp /etc/group2.tmp
fi
```

No surprises there. Continuing our uninstall, we start on the network-related tasks:

```
# Next, lose the amanda line in /etc/services...
# We only want to do this if the server package isn't installed
# Look for /usr/sbin/amdump, and leave it if there...

if [ ! -f /usr/sbin/amdump ];
then

        if grep "^amanda" /etc/services > /dev/null
        then
                grep -v "^amanda" /etc/services > /etc/services.tmp
                mv -f /etc/services.tmp /etc/services
        fi
fi
```

That's odd. Why are we looking for a file from the server package? If you look back at the install scripts for the client and server packages, you'll find that the one thing they have in common is that both the client and the server require the same entry in `/etc/services`.

If an amanda server is going to back itself up, it also needs the amanda client software. Therefore, both subpackages need to add an entry to `/etc/services`. But what if one of the packages is removed? Perhaps the server is being demoted to a client, or maybe the server is no longer going to be backed up using amanda. In these cases, the entry in`/etc/services` must stay. So, in the case of the client, we look for a file from the server subpackage, and if it's there, we leave the entry alone.

Granted, this is a somewhat unsightly way to see if a certain package is installed. Some of you are probably even saying, "Why can't RPM be used? Just do an `rpm -q amanda-server`, and decide what to do based on that." And that would be the best way to do it, except for one small point:

Only one invocation of RPM can run at any given time.

Since RPM is running to perform the uninstall, if the uninstall-script were to attempt to run RPM again, it would fail. The reason it would fail is because only one copy of RPM can access the database at a time. So we are stuck with our unsightly friend.

Continuing the network-related uninstall tasks:

```
# Finally, the amanda entry in /etc/inetd.conf

if grep "^amanda" /etc/inetd.conf > /dev/null
then
        grep -v "^amanda" /etc/inetd.conf > /etc/inetd.conf.tmp
        mv -f /etc/inetd.conf.tmp /etc/inetd.conf

# Kick inetd

if [ -f /var/run/inetd.pid ];
```

```
then
        kill -HUP `cat /var/run/inetd.pid`
fi
fi

fi
```

Here, we're using `grep`'s ability to return lines that *don't* match the search string, in order to remove every trace of amanda from `/etc/inetd.conf`. After issuing a hup on inetd, we're done.

On to the server. If you've been noticing a pattern between the various scripts, you won't be disappointed here:

```
%postun server

# See if we're the last server package on the system...
# If not, we don't need to do any of this stuff...

if [ "$1" = 0 ];
then

# Lose the amanda line in /etc/services...
# We only want to do this if the client package isn't installed
# Look for /usr/lib/amandad, and leave it if there...

if [ ! -f /usr/lib/amanda/amandad ];
then

        if grep "^amanda" /etc/services > /dev/null
        then
                grep -v "^amanda" /etc/services > /etc/services.tmp
                mv -f /etc/services.tmp /etc/services
        fi
fi

fi
```

By now the opening `if` statement is an old friend. As you might have expected, we are verifying whether the client package is installed, by looking for a file from that package. If the client package isn't there, the entry is removed from `/etc/services`. And that, is that.

Obviously, these scripts must be carefully tested. In the case of amanda, since the two subpackages have some measure of interdependency, it's necessary to try different sequences of installing and erasing the two packages to make sure the `/etc/services` logic works properly in all cases.

After a bit of testing, our install and uninstall scripts pass with flying colors. From a technological standpoint, the client and server subpackages are ready to go.

Bits and Pieces

However, just because a package has been properly built, and installs and can be erased without problems, doesn't mean that the package builder's job is done. It's necessary to look at each newly-built package from the user's perspective. Does the package contain everything the user needs in order to deploy it effectively? Or will the user need to fiddle with it, guessing as they go?

In the case of our amanda packages, it was obvious that some additional documentation was required so that the user would know what needed to be done in order to finalize the installation. Simply directing the user to the standard amanda documentation wasn't the right solution, either. Many of the steps outlined in the `INSTALL` document had already been done by the post-install scripts. No, an interim documente was required. Two, actually: one for the client, and one for the server.

So two files were created, one to be added to each subpackage. The question was, how to do it? Essentially, there were two options:

1. Put the files in the amanda directory tree that had been used to perform the initial builds and generate a new patch file
2. Create a `tar` file containing the two files, and modify the spec file to unpack the documentation into the amanda directory tree.
3. Drop the files directly into the amanda directory tree without using `tar`.

Since the second approach was more interesting, that's the approach we chose. It required an additional `source` tag in the spec file:

```
Source1: amanda-rpm-instructions.tar.gz
```

Also required was an additional `%setup` macro in the `%prep` script:

```
%setup -T -D -a 1
```

While the `%setup` macro might look intimidating, it wasn't that hard to construct. Here's what each options means:

`-T` — Do not perform the default archive unpacking.

`-D` — Do not delete the directory before unpacking.

`-a 1` — Unpack the archive specified by the `source1` tag after changing directory.

Finally, two additions to the `%files` lists were required. One for the client:

```
%doc amanda-client.README
```

And one for the server:

```
%doc amanda-server.README
```

At this point, the packages were complete. Certainly there is software out there that doesn't require this level of effort to package. Just as certainly there is software that is much more of a challenge. Hopefully this chapter has given you some idea about how to approach package building for more complex applications.

Chapter 21

A Guide to the RPM Library API

In this chapter, we'll explore the functions used internally by RPM. These functions are available for anyone to use, making it possible to add RPM functionality to new and existing programs. Rather than continually refer to "the RPM library" throughout this chapter, we'll use the name of the library's main include file — rpmlib.

21.1 An Overview of rpmlib

There are a number of files that make up rpmlib. First and foremost, of course, is the rpmlib library, `librpm.a`. This library contains all the functions required to implement all the basic functions contained in RPM.

The remaining files define the various data structures, parameters, and symbols used by rpmlib:

- `rpmlib.h`
- `dbindex.h`
- `header.h`

In general, `rpmlib.h` will always be required. When using rpmlib's header-related functions, `header.h` will be required, while the database-related function will require `dbindex.h`. As each function is described in this chapter, we'll provide the function's prototype as well as the `#include` statements the function requires.

21.2 rpmlib Functions

There are more than sixty different functions in rpmlib. The tasks they perform range from low-level database record traversal, to high-level package manipulation. We've grouped the functions into different categories for easy reference.

21.2.1 Error Handling

The functions in this section perform rpmlib's basic error handling. All error handling centers on the use of specific status codes. The status codes are defined in `rpmlib.h` and are of the form `RPMERR_`*xxx*, where *xxx* is the name of the error.

Return Error Code — `rpmErrorCode()`

```
#include <rpm/rpmlib.h>

int rpmErrorCode(void);
```

This function returns the error code set by the last rpmlib function that failed. Should only be used in an error callback function defined by `rpmErrorSetCallBack()`.

Return Error String — `rpmErrorString()`

```
#include <rpm/rpmlib.h>

char *rpmErrorString(void);
```

This function returns the error string set by the last rpmlib function that failed. Should only be used in an error callback function defined by `rpmErrorSetCallBack()`.

Set Error CallBack Function — `rpmErrorSetCallback()`

```
#include <rpm/rpmlib.h>

rpmErrorCallBackType rpmErrorSetCallback(rpmErrorCallBackType);
```

This function sets the current error callback function to the error callback function passed to it. The previous error callback function is returned.

21.2.2 Getting Package Information

The following functions are used to obtain information about a package file.

It should be noted that most information is returned in the form of a `Header` structure. This data structure is widely used throughout rpmlib. We will discuss more header-related functions in sections 21.2.13 and 21.2.14 starting at page 332.

Read Package Information — `rpmReadPackageInfo()`

```
#include <rpm/rpmlib.h>
#include <rpm/header.h>

int rpmReadPackageInfo(int fd,
                       Header * signatures,
                       Header * hdr);
```

Given an open package on `fd`, read in the header and signature. This function operates as expected with both socket and pipe file descriptors passed as `fd`. Safe on nonseekable `fd`s. When the function returns, `fd` is left positioned at the start of the package's archive section.

If either `signatures` or `hdr` are `NULL`, information for the `NULL` parameter will not be passed back to the caller. Otherwise, they will return the package's signatures and header, respectively.

This function returns the following status values:

- `0` — Success.
- `1` — Bad magic numbers found in package.
- `2` — Other error.

Read Package Header — `rpmReadPackageHeader()`

```
#include <rpm/rpmlib.h>
#include <rpm/header.h>

int rpmReadPackageHeader(int fd,
                         Header * hdr,
                         int * isSource,
                         int * major,
                         int * minor);
```

Given an open package on `fd`, read in the header. This function operates as expected with both socket and pipe file descriptors passed as `fd`. Safe on nonseekable `fd`s. When the function returns, `fd` is left positioned at the start of the package's archive section.

If `hdr`, `isSource`, `major`, or `minor` are `NULL`, information for the `NULL` parameter(s) will not be passed back to the caller. Otherwise, they will return the package's header (`hdr`), information on whether the package is a source package file or not (`isSource`), and the package format's major and minor revision number (`major` and `minor`, respectively).

This function returns the following status values:

0 — Success.

1 — Bad magic numbers found in package.

2 — Other error.

21.2.3 Variable Manipulation

The following functions are used to get, set, and interpret RPM's internal variables. Variables are set according to various pieces of system information, as well as from `rpmrc` files. They control various aspects of RPM's operation.

The variables have symbolic names in the form `RPMVAR_`*xxx*, where *xxx* is the name of the variable. All variable names are defined in `rpmlib.h`.

Return Value of RPM Variable — `rpmGetVar()`

```
#include <rpm/rpmlib.h>

char *rpmGetVar(int var);
```

This function returns the value of the variable specified in `var`.

On error, the function returns `NULL`.

Return Boolean Value Of RPM Variable — `rpmGetBooleanVar()`

```
#include <rpm/rpmlib.h>

int rpmGetBooleanVar(int var);
```

This function looks up the variable specified in `var` and returns a 0 or 1 depending on the variable's value.

On error, the function returns 0.

Set Value Of RPM Variable — `rpmSetVar()`

```
#include <rpm/rpmlib.h>

void rpmSetVar(int var,
               char *val);
```

This function sets the variable specified in `var` to the value passed in `val`. It is also possible for `val` to be `NULL`.

21.2.4 `rpmrc`-Related Information

The functions in this section are all related to `rpmrc` information — the `rpmrc` files as well as the variables set from those files. This information also includes the architecture and operating system information based on `rpmrc` file entries.

Read `rpmrc` Files — `rpmReadConfigFiles()`

```
#include <rpm/rpmlib.h>

int rpmReadConfigFiles(char * file,
                       char * arch,
                       char * os,
                       int building);
```

This function reads `rpmrc` files according to the following rules:

- Always read `/usr/lib/rpmrc`.
- If `file` is specified, read it.
- If `file` is not specified, read `/etc/rpmrc` and `~/.rpmrc`.

Every `rpmrc` file entry is used with `rpmSetVar()` to set the appropriate RPM variable. Part of the normal `rpmrc` file processing also includes setting the architecture and operating system variables for the system executing this function. These default settings can be overridden by entering architecture and/or operating system information in `arch` and `os`, respectively. This information will still go through the normal `rpmrc` translation process.

The `building` argument should be set to 1 only if a package is being built when this function is called. Since most rpmlib-based applications will probably not duplicate RPM's package building capabilities, `building` should normally be set to 0.

Return Operating System Name — `rpmGetOsName()`

```
#include <rpm/rpmlib.h>

char *rpmGetOsName(void);
```

This function returns the name of the operating system, as determined by rpmlib's normal `rpmrc` file processing.

Return Architecture Name — `rpmGetArchName()`

```
#include <rpm/rpmlib.h>

char *rpmGetArchName(void);
```

This function returns the name of the architecture, as determined by rpmlib's normal `rpmrc` file processing.

Print all `rpmrc`-Derived Variables — `rpmShowRC()`

```
#include <rpm/rpmlib.h>

int rpmShowRC(FILE *f);
```

This function writes all variable names and their values to the file `f`. Always returns 0.

Return Architecture Compatibility Score — `rpmArchScore()`

```
#include <rpm/rpmlib.h>

int rpmArchScore(char * arch);
```

This function returns the "distance" between the architecture whose name is specified in `arch`, and the current architecture. Returns 0 if the two architectures are incompatible. The smaller the number returned, the more compatible the two architectures are.

Return Operating System Compatibility Score — `rpmOsScore()`

```
#include <rpm/rpmlib.h>

int rpmOsScore(char * os);
```

This function returns the "distance" between the operating system whose name is specified in `os`, and the current operating system. Returns 0 if the two operating systems are incompatible. The smaller the number returned, the more compatible the two operating systems are.

21.2.5 RPM Database Manipulation

The functions in this section perform the basic operations on the RPM database. This includes opening and closing the database, as well as creating the database. A function also exists to rebuild a database that has been corrupted.

Every function that accesses the RPM database in some fashion makes use of the `rpmdb` structure. This structure is used as a handle to refer to a particular RPM database.

Open RPM Database — `rpmdbOpen()`

```
#include <rpm/rpmlib.h>
```

```
int rpmdbOpen(char * root,
              rpmdb * dbp,
              int mode,
              int perms);
```

This function opens the RPM database located in RPMVAR_DBPATH, returning the rpmdb structure dbp. If root is specified, it is prepended to RPMVAR_DBPATH prior to opening. The mode and perms parameters are identical to open(2)'s flags and mode parameters, respectively.

The function returns 1 on error, 0 on success.

Close RPM Database — rpmdbClose()

```
#include <rpm/rpmlib.h>

void rpmdbClose(rpmdb db);
```

This function closes the RPM database specified by the rpmdb structure db. The db structure is also freed.

Create RPM Database — rpmdbInit()

```
#include <rpm/rpmlib.h>

int rpmdbInit(char * root,
              int perms);
```

This function creates a new RPM database to be located in RPMVAR_DBPATH. If the database already exists, it is left unchanged. If root is specified, it is prepended to RPMVAR_DBPATH prior to creation. The perms parameter is identical to open(2)'s mode parameter.

The function returns 1 on error, 0 on success.

Rebuild RPM Database — rpmdbRebuild()

```
#include <rpm/rpmlib.h>

int rpmdbRebuild(char * root);
```

This function rebuilds the RPM database located in RPMVAR_DBPATH. If root is specified, it is prepended to RPMVAR_DBPATH prior to rebuilding.

The function returns 1 on error, 0 on success.

21.2.6 RPM Database Traversal

The following functions are used to traverse the RPM database. Also described in this section is a function to retrieve a database record by its record number.

It should be noted that database records are returned in the form of a `Header` structure. This data structure is widely used throughout rpmlib. We will discuss more header-related functions in sections 21.2.13 and 21.2.14 starting at page 332.

Begin RPM Database Traversal — `rpmdbFirstRecNum()`

```
#include <rpm/rpmlib.h>

unsigned int rpmdbFirstRecNum(rpmdb db);
```

This function returns the record number of the first record in the database specified by `db`.

On error, it returns 0.

Traverse To Next RPM Database Record — `rpmdbNextRecNum()`

```
#include <rpm/rpmlib.h>

unsigned int rpmdbNextRecNum(rpmdb db,
                             unsigned int lastOffset);
```

This function returns the record number of the record following the record number passed in `lastOffset`, in the database specified by `db`.

On error, this function returns 0.

Return Record From RPM Database — `rpmdbGetRecord()`

```
#include <rpm/rpmlib.h>

Header rpmdbGetRecord(rpmdb db,
                      unsigned int offset);
```

This function returns the record at the record number specified by `offset` from the database specified by `db`.

This function returns `NULL` on error.

21.2.7 RPM Database Search

The functions in this section search the various parts of the RPM database. They all return a structure of type `dbiIndexSet`, which contains the records that match the search term. Here is the definition of the structure, as found in `<rpm/dbindex.h>`:

```
typedef struct {
    dbiIndexRecord * recs;
    int count;
} dbiIndexSet;
```

Each `dbiIndexRecord` is also defined in `<rpm/dbindex.h>` as follows:

```
typedef struct {
    unsigned int recOffset;
    unsigned int fileNumber;
} dbiIndexRecord;
```

The `recOffset` element is the offset of the record from the start of the database file. The `fileNumber` element is only used by `rpmdbFindByFile()`.

Keep in mind that the `rpmdbFind`*xxx* search functions each return `dbiIndexSet` structures, which must be freed with `dbiFreeIndexRecord()` when no longer needed.

Free Database Index — `dbiFreeIndexRecord()`

```
#include <rpm/rpmlib.h>
#include <rpm/dbindex.h>

void dbiFreeIndexRecord(dbiIndexSet set);
```

This function frees the database index set specified by `set`.

Search RPM Database By File — `rpmdbFindByFile()`

```
#include <rpm/rpmlib.h>
#include <rpm/dbindex.h>

int rpmdbFindByFile(rpmdb db,
                    char * filespec,
                    dbiIndexSet * matches);
```

This function searches the RPM database specified by `db` for the package which owns the file specified by `filespec`. It returns matching records in `matches`.

This function returns the following status values:

- `-1` — An error occurred reading a database record.
- `0` — The search completed normally.
- `1` — The search term was not found.

Search RPM Database By Group — `rpmdbFindByGroup()`

```
#include <rpm/rpmlib.h>
#include <rpm/dbindex.h>

int rpmdbFindByGroup(rpmdb db,
                     char * group,
                     dbiIndexSet * matches);
```

This function searches the RPM database specified by `db` for the packages which are members of the group specified by `group`. It returns matching records in `matches`.

This function returns the following status values:

- `-1` — An error occurred reading a database record.
- `0` — The search completed normally.
- `1` — The search term was not found.

Search RPM Database By Package — `rpmdbFindPackage()`

```
#include <rpm/rpmlib.h>
#include <rpm/dbindex.h>

int rpmdbFindPackage(rpmdb db,
                     char * name,
                     dbiIndexSet * matches);
```

This function searches the RPM database specified by `db` for the packages with the package name (not label) specified by `name`. It returns matching records in `matches`.

This function returns the following status values:

- `-1` — An error occurred reading a database record.
- `0` — The search completed normally.
- `1` — The search term was not found.

Search RPM Database By Provides — `rpmdbFindByProvides()`

```
#include <rpm/rpmlib.h>
#include <rpm/dbindex.h>

int rpmdbFindByProvides(rpmdb db,
                        char * provides,
                        dbiIndexSet * matches);
```

This function searches the RPM database specified by `db` for the packages which provide the provides information specified by `provides`. It returns matching records in `matches`.

This function returns the following status values:

- `-1` — An error occurred reading a database record.
- `0` — The search completed normally.
- `1` — The search term was not found.

Search RPM Database By Requires — `rpmdbFindByRequiredBy()`

```
#include <rpm/rpmlib.h>
#include <rpm/dbindex.h>

int rpmdbFindByRequiredBy(rpmdb db,
                          char * requires,
                          dbiIndexSet * matches);
```

This function searches the RPM database specified by `db` for the packages which require the requires information specified by `requires`. It returns matching records in `matches`.

This function returns the following status values:

- `-1` — An error occurred reading a database record.
- `0` — The search completed normally.
- `1` — The search term was not found.

Search RPM Database By Conflicts — `rpmdbFindByConflicts()`

```
#include <rpm/rpmlib.h>
#include <rpm/dbindex.h>

int rpmdbFindByConflicts(rpmdb db,
                         char * conflicts,
                         dbiIndexSet * matches);
```

This function searches the RPM database specified by `db` for the packages which conflict with the conflicts information specified by `conflicts`. It returns matching records in `matches`.

This function returns the following status values:

- `-1` — An error occurred reading a database record.
- `0` — The search completed normally.
- `1` — The search term was not found.

21.2.8 Package Manipulation

These functions perform the operations most RPM users are familiar with. Functions that install and erase packages are here, along with a few related lower-level support functions.

Install Source Package File — `rpmInstallSourcePackage()`

```
#include <rpm/rpmlib.h>

int rpmInstallSourcePackage(char * root,
                            int fd,
                            char ** specFile,
                            rpmNotifyFunction notify,
                            char * labelFormat);
```

This function installs the source package file specified by `fd`. If `root` is not `NULL`, it is prepended to the variables `RPMVAR_SOURCEDIR` and `RPMVAR_SPECDIR` prior to the actual installation. If `specFile` is not `NULL`, the complete path and filename of the just-installed spec file is returned.

The `notify` parameter is used to specify a progress-tracking function that will be called during the installation. Please refer to page 326 for more information on this parameter.

The `labelFormat` parameter can be used to specify how the package label should be formatted. It is used when printing the package label once the package install is ready to proceed. If `labelformat` is `NULL`, the package label is not printed.

This function returns the following status values:

- `0` — The source package was installed successfully.
- `1` — The source package file contained incorrect magic numbers.
- `2` — Another type of error occurred.

Install Binary Package File — `rpmInstallPackage()`

```
#include <rpm/rpmlib.h>

int rpmInstallPackage(char * rootdir,
                      rpmdb db,
                      int fd,
                      char * prefix,
                      int flags,
                      rpmNotifyFunction notify,
                      char * labelFormat,
                      char * netsharedPath);
```

This function installs the binary package specified by `fd`. If a path is specified in `rootdir`, the package will be installed with that path acting as the root directory. If a path is specified in `prefix`, it will be used as the prefix for relocatable packages. The RPM database specified by `db` is updated to reflect the newly installed package.

The `flags` parameter is used to control the installation behavior. The flags are defined in `rpmlib.h` and take the form `RPMINSTALL_`*xxx*, where *xxx* is the name of the flag.

The following flags are currently defined:

- `RPMINSTALL_REPLACEPKG` — Install the package even if it is already installed.
- `RPMINSTALL_REPLACEFILES` — Install the package even if it will replace files owned by another package.
- `RPMINSTALL_TEST` — Perform all install-time checks, but do not actually install the package.
- `RPMINSTALL_UPGRADE` — Install the package, and remove all older versions of the package.
- `RPMINSTALL_UPGRADETOOLD` — Install the package, even if the package is an older version of an already-installed package.
- `RPMINSTALL_NODOCS` — Do not install the package's documentation files.
- `RPMINSTALL_NOSCRIPTS` — Do not execute the package's install- and erase-time (in the case of an upgrade) scripts.
- `RPMINSTALL_NOARCH` — Do not perform architecture compatibility tests.
- `RPMINSTALL_NOOS` — Do not perform operating system compatibility tests.

The `notify` parameter is used to specify a progress tracking function that will be called during the installation. Please refer to page 326 for more information on this parameter.

The `labelFormat` parameter can be used to specify how the package label should be formatted. This information is used when printing the package label once the package install is ready to proceed. It is used when printing the package label once the package install is ready to proceed. If `labelformat` is `NULL`, the package label is not printed.

The `netsharedPath` parameter is used to specify that part of the local filesystem that is shared with other systems. If there is more than one path that is shared, the paths should be separated with a colon.

This function returns the following status values:

`0` — The binary package was installed successfully.

`1` — The binary package file contained incorrect magic numbers.

`2` — Another type of error occurred.

Track Package Installation Progress — `rpmNotifyFunction()`

```
#include <rpm/rpmlib.h>

typedef void (*rpmNotifyFunction)(const unsigned long amount,
                                  const unsigned long total);
```

A function can be passed to `rpmIstallSourcePackage` or `rpmInstallPackage` via the `notify` parameter. The function will be called at regular intervals during the installation, and will have two parameters passed to it:

1. `amount` — The number of bytes of the install that have been completed so far.
2. `total` — The total number of bytes that will be installed.

This function permits the creation of a dynamically updating progress meter during package installation.

Remove Installed Package — `rpmRemovePackage()`

```
#include <rpm/rpmlib.h>

int rpmRemovePackage(char * root,
                     rpmdb db,
                     unsigned int offset,
                     int flags);
```

This function removes the package at record number `offset` in the RPM database specified by `db`. If `root` is specified, it is used as the path to a directory that will serve as the root directory while the package is being removed.

The `flags` parameter is used to control the package removal behavior. The flags that may be passed are defined in `rpmlib.h`, and are of the form `RPMUNINSTALL_`***xxx***, where ***xxx*** is the name of the flag.

The following flags are currently defined:

- `RPMUNINSTALL_TEST` — Perform all erase-time checks, but do not actually remove the package.
- `RPMUNINSTALL_NOSCRIPTS` — Do not execute the package's erase-time scripts.

This function returns the following status values:

`0` — The package was removed successfully.

`1` — The package removal failed.

21.2.9 Package And File Verification

The functions in this section perform the verification operations necessary to ensure that the files comprising a package have not been modified since they were installed.

Verification takes place on three distinct levels:

1. On the file-by-file level.
2. On a package-wide level, through the use of the `%verifyscript` verification script.
3. On an inter-package level, through RPM's normal dependency processing.

Because of this, there are two functions to perform each specific verification operation.

Verify File — `rpmVerifyFile()`

```
#include <rpm/rpmlib.h>
#include <rpm/header.h>

int rpmVerifyFile(char * root,
                  Header h,
                  int filenum,
                  int * result);
```

This function verifies the `filenum`'th file from the package whose header is `h`. If `root` is specified, it is used as the path to a directory that will serve as the root directory while the file is being verified. The results of the file verification are returned in `result`, and consist of a number of flags. Each flag that is set indicates a verification failure.

The flags are defined in `rpmlib.h`, and are of the form `RPMVERIFY_`*xxx*, where *xxx* is the name of the data that failed verification.

This function returns `0` on success, and `1` on failure.

Execute Package's `%verifyscript` Verification Script — `rpmVerifyScript()`

```
#include <rpm/rpmlib.h>
#include <rpm/header.h>

int rpmVerifyScript(char * root,
                    Header h,
                    int err);
```

This function executes the `%verifyscript` verification script for the package whose header is `h`. `err` must contain a valid file descriptor. If `rpmIsVerbose()` returns true, the `%verifyscript` verification script will direct all status messages to `err`.

This function returns `0` on success, `1` on failure.

21.2.10 Dependency-Related Operations

The functions in this section are used to perform the various dependency-related operations supported by rpmlib.

Dependency processing is entirely separate from normal package-based operations. The package installation and removal functions do not perform any dependency processing themselves. Therefore, dependency processing is somewhat different from other aspects of rpmlib's operation.

Dependency processing centers around the `rpmDependencies` data structure. The operations that are to be performed against the RPM database (adding, removing, and upgrading packages) are performed against this data structure, using functions that are described below. These functions simply populate the data structure according to the operation being performed. They do *not* perform the actual operation on the package. This is an important point to keep in mind.

Once the data structure has been completely populated, a dependency check function is called to determine if there are any dependency-related problems. The result is a structure of dependency conflicts. This structure, `rpmDependencyConflict`, is defined in `rpmlib.h`.

Note that it is necessary to free both the conflicts structure *and* the `rpmDependencies` structure when they are no longer needed. However, `free()` should *not* be used — special functions for this are provided, and will be discussed in this section.

Create a New Dependency Data Structure — `rpmdepDependencies()`

```
#include <rpm/rpmlib.h>

rpmDependencies rpmdepDependencies(rpmdb db);
```

This function returns an initialized `rpmDependencies` structure. The dependency checking to be done will be based on the RPM database specified in the `db` parameter. If this parameter is `NULL`, the dependency checking will be done as if an empty RPM database was being used.

Add a Package Install To the Dependency Data Structure — `rpmdepAddPackage()`

```
#include <rpm/rpmlib.h>
#include <rpm/header.h>

void rpmdepAddPackage(rpmDependencies rpmdep,
                      Header h);
```

This function adds the installation of the package whose header is `h`, to the `rpmDependencies` data structure, `rpmdep`.

Add a Package Upgrade To the Dependency Data Structure — `rpmdepUpgradePackage()`

```
#include <rpm/rpmlib.h>
#include <rpm/header.h>

void rpmdepUpgradePackage(rpmDependencies rpmdep,
                          Header h);
```

This function adds the upgrading of the package whose header is `h`, to the `rpmDependencies` data structure, `rpmdep`. It is similar to `rpmdepAddPackage()`, but older versions of the package are removed.

Add a Package Removal To the Dependency Data Structure — `rpmdepRemovePackage()`

```
#include <rpm/rpmlib.h>

void rpmdepRemovePackage(rpmDependencies rpmdep,
                         int dboffset);
```

This function adds the removal of the package whose RPM database offset is `dboffset`, to the `rpmDependencies` data structure, `rpmdep`.

Add an Available Package To the Dependency Data Structure — `rpmdepAvailablePackage()`

```
#include <rpm/rpmlib.h>
#include <rpm/header.h>

void rpmdepAvailablePackage(rpmDependencies rpmdep,
                            Header h,
                            void * key);
```

This function adds the package whose header is `h`, to the `rpmDependencies` structure, `rpmdep`.

The `key` parameter can be anything that uniquely identifies the package being added. It will be returned as part of the `rpmDependencyConflict` structure returned by `rpmdepCheck()`, specifically in that structure's `suggestedPackage` element.

Perform a Dependency Check — `rpmdepCheck()`

```
#include <rpm/rpmlib.h>

int rpmdepCheck(rpmDependencies rpmdep,
```

```
                struct rpmDependencyConflict ** conflicts,
                int * numConflicts);
```

This function performs a dependency check on the `rpmDependencies` structure `rpmdep`. It returns an array of size `numConflicts`, pointed to by `conflicts`.

This function returns `0` on success, and `1` on error.

Free Results of `rpmdepCheck()` — `rpmdepFreeConflicts()`

```
#include <rpm/rpmlib.h>

void rpmdepFreeConflicts(struct rpmDependencyConflict * conflicts,
                         int numConflicts);
```

This function frees the dependency conflict information of size `numConflicts` pointed to by `conflicts`.

Free a Dependency Data Structure — `rpmdepDone()`

```
#include <rpm/rpmlib.h>

void rpmdepDone(rpmDependencies rpmdep);
```

This function frees the `rpmDependencies` structure pointed to by `rpmdep`.

21.2.11 Diagnostic Output Control

The functions in this section are used to control the amount of diagnostic output produced by other rpmlib functions. The rpmlib library can produce a wealth of diagnostic output, making it easy to see what is going on at any given time.

There are several different verbosity levels defined in `rpmlib.h`. Their symbolic names are of the form `RPMMESS_`*xxx*, where *xxx* is the name of the verbosity level. It should be noted that the numeric values of the verbosity levels *increase* with a *decrease* in verbosity.

Unless otherwise set, the default verbosity level is `RPMMESS_NORMAL`.

Increase Verbosity Level — `rpmIncreaseVerbosity()`

```
#include <rpm/rpmlib.h>

void rpmIncreaseVerbosity(void);
```

This function is used to increase the current verbosity level by one.

Set Verbosity Level — `rpmSetVerbosity()`

```
#include <rpm/rpmlib.h>
```

```
void rpmSetVerbosity(int level);
```

This function is used to set the current verbosity level to `level`. Note that no range checking is done to `level`.

Return Verbosity Level — `rpmGetVerbosity()`

```
#include <rpm/rpmlib.h>
```

```
int rpmGetVerbosity(void);
```

This function returns the current verbosity level.

Check Verbosity Level — `rpmIsVerbose()`

```
#include <rpm/rpmlib.h>
```

```
int rpmIsVerbose(void);
```

This function checks the current verbosity level and returns `1` if the current level is set to `RPMMESS_VERBOSE` or a level of higher verbosity. Otherwise, it returns `0`.

Check Debug Level — `rpmIsDebug()`

```
#include <rpm/rpmlib.h>
```

```
int rpmIsDebug(void);
```

This function checks the current verbosity level and returns `1` if the current level is set to `RPMMESS_DEBUG`, or a level of higher verbosity. Otherwise, it returns `0`.

21.2.12 Signature Verification

The functions in this section deal with the verification of package signatures. A package file may contain more than one type of signature. For example, a package may contain a signature that contains the package's size, as well as a signature that contains cryptographically-derived data that can be used to prove the package's origin.

Each type of signature has its own tag value. These tag values are defined in `rpmlib.h` and are of the form `RPMSIGTAG_`*xxx*, where *xxx* is the type of signature.

Verify A Package File's Signature — `rpmVerifySignature()`

```
#include <rpm/rpmlib.h>

int rpmVerifySignature(char *file,
                       int_32 sigTag,
                       void *sig,
                       int count,
                       char *result);
```

This function verifies the signature of the package pointed to by `file`. The result of the verification is stored in `result`, in a format suitable for printing.

The `sigTag` parameter specifies the type of signature to be checked. The `sig` parameter specifies the signature against which the package is to be verified. The `count` parameter specifies the size of the signature; at present, this parameter is only used for PGP-based signatures.

This function returns the following values:

- `RPMSIG_OK` — The signature verified correctly.
- `RPMSIG_UNKNOWN` — The signature type is unknown.
- `RPMSIG_BAD` — The signature did not verify correctly.
- `RPMSIG_NOKEY` — The key required to check this signature is not available.

Free Signature Read By `rpmReadPackageInfo()` — `rpmFreeSignature()`

```
#include <rpm/rpmlib.h>
#include <rpm/header.h>

void rpmFreeSignature(Header h);
```

This function frees the signature `h`.

21.2.13 Header Manipulation

The header is one of the key data structures in rpmlib. The functions in this section perform basic manipulations of the header.

The header is actually a data structure. It is not necessary to fully understand the actual data structure. However, it *is* necessary to understand the basic concepts on which the header is based.

The header serves as a kind of miniature database. The header can be searched for specific information, which can be retrieved easily. Like a database, the information contained in the header can be of varying sizes.

Read A Header — `headerRead()`

```
#include <rpm/rpmlib.h>
#include <rpm/header.h>

Header headerRead(int fd,
                  int magicp);
```

This function reads a header from file `fd`, converting it from network byte order to the host system's byte order. If `magicp` is defined to be `HEADER_MAGIC_YES`, `headerRead()` will expect header magic numbers, and will return an error if they are not present. Likewise, if `magicp` is defined to be `HEADER_MAGIC_NO`, `headerRead()` will not check the header's magic numbers, and will return an error if they are present.

On error, this function returns `NULL`.

Write A Header — `headerWrite()`

```
#include <rpm/rpmlib.h>
#include <rpm/header.h>

void headerWrite(int fd,
                 Header h,
                 int magicp);
```

This function writes the header `h`, to file `fd`, converting it from host byte order to network byte order. If `magicp` is defined to be `HEADER_MAGIC_YES`, `headerWrite()` will add the appropriate magic numbers to the header being written. If `magicp` is defined to be `HEADER_MAGIC_NO`, `headerWrite()` will not include magic numbers.

Copy A Header — `headerCopy()`

```
#include <rpm/rpmlib.h>
#include <rpm/header.h>

Header headerCopy(Header h);
```

This function returns a copy of header `h`.

Calculate A Header's Size — `headerSizeof()`

```
#include <rpm/rpmlib.h>
#include <rpm/header.h>

unsigned int headerSizeof(Header h,
                          int magicp);
```

This function returns the number of bytes the header `h` takes up on disk. Note that in versions of RPM prior to 2.3.3, this function also changes the location of the data in the header. The result is that pointers from `headerGetEntry()` will no longer be valid. Therefore, any pointers acquired before calling `headerSizeof()` should be discarded.

Create A New Header — `headerNew()`

```
#include <rpm/rpmlib.h>
#include <rpm/header.h>

Header headerNew(void);
```

This function returns a new header.

Deallocate A Header — `headerFree()`

```
#include <rpm/rpmlib.h>
#include <rpm/header.h>

void headerFree(Header h);
```

This function deallocates the header specified by `h`.

Print Header Structure In Human-Readable Form — `headerDump()`

```
#include <rpm/rpmlib.h>
#include <rpm/header.h>

void headerDump(Header h,
                FILE *f,
                int flags);
```

This function prints the structure of the header `h`, to the file `f`. If the `flags` parameter is defined to be `HEADER_DUMP_INLINE`, the header's data is also printed.

21.2.14 Header Entry Manipulation

The functions in this section provide the basic operations necessary to manipulate header entries. The following header entry types are currently defined:

- `RPM_NULL_TYPE` — This type is not used.
- `RPM_CHAR_TYPE` — The entry contains a single character.
- `RPM_INT8_TYPE` — The entry contains an eight-bit integer.

- `RPM_INT16_TYPE` — The entry contains a sixteen-bit integer.
- `RPM_INT32_TYPE` — The entry contains a thirty-two-bit integer.
- `RPM_INT64_TYPE` — The entry contains a sixty-four-bit integer.
- `RPM_STRING_TYPE` — The entry contains a null-terminated character string.
- `RPM_BIN_TYPE` — The entry contains binary data that will not be interpreted by rpmlib.
- `RPM_STRING_ARRAY_TYPE` — The entry contains an array of null-terminated strings.

Get Entry From Header — `headerGetEntry()`

```
#include <rpm/rpmlib.h>
#include <rpm/header.h>

int headerGetEntry(Header h,
                   int_32 tag,
                   int_32 *type,
                   void **p,
                   int_32 *c);
```

This function retrieves the entry matching `tag` from header `h`. The type of the entry is returned in `type`, a pointer to the data is returned in `p`, and the size of the data is returned in `c`. Both `type` and `c` may be null, in which case that data will not be returned. Note that if the entry type is `RPM_STRING_ARRAY_TYPE`, you must issue a `free()` on `p` when done with the data.

This function returns `1` on sucess, and `0` on failure.

Add Entry To Header — `headerAddEntry()`

```
#include <rpm/rpmlib.h>
#include <rpm/header.h>

int headerAddEntry(Header h,
                   int_32 tag,
                   int_32 type,
                   void *p,
                   int_32 c);
```

This function adds a new entry to the header `h`. The entry's tag is specified by the `tag` parameter, and the entry's type is specified by the `type` parameter.

The entry's data is pointed to by `p`, and the size of the data is specified by `c`.

This function always returns `1`.

Note: In versions of RPM prior to 2.3.3, `headerAddEntry()` will only work successfully with headers produced by `headerCopy()` and `headerNew()`. In particular, `headerAddEntry()` is not supported when used to add entries to a header produced by `headerRead()`. Later versions of RPM lift this restriction.

Determine If Entry Is In Header — `headerIsEntry()`

```
#include <rpm/rpmlib.h>
#include <rpm/header.h>

int headerIsEntry(Header h,
                  int_32 tag);
```

This function returns `1` if an entry with tag `tag` is present in header `h`. If the tag is not present, this function returns `0`.

21.2.15 Header Iterator Support

Iterators are used as a means to step from entry to entry, through an entire header. The functions in this section are used to create, use, and free iterators.

Create an Iterator — `headerInitIterator()`

```
#include <rpm/rpmlib.h>
#include <rpm/header.h>

HeaderIterator headerInitIterator(Header h);
```

This function returns a newly-created iterator for the header `h`.

Step To the Next Entry — `headerNextIterator()`

```
#include <rpm/rpmlib.h>
#include <rpm/header.h>

int headerNextIterator(HeaderIterator iter,
                       int_32 *tag,
                       int_32 *type,
                       void **p,
                       int_32 *c);
```

This function steps to the next entry in the header specified when the iterator `iter` was created with `headerInitIterator()`. The next entry's tag, type, data, and size are returned in `tag`, `type`, `p`, and `c`, respectively. Note that if the entry type is `RPM_STRING_ARRAY_TYPE`, you must issue a `free()` on `p` when done with the data.

This function returns `1` if successful, and `0` if there are no more entries in the header.

Free An Iterator — `headerFreeIterator()`

```
#include <rpm/rpmlib.h>
#include <rpm/header.h>

void headerFreeIterator(HeaderIterator iter);
```

This function frees the resources used by the iterator `iter`.

21.3 Example Code

In this section, we'll study example programs that make use of rpmlib to perform an assortment of commonly-required operations.

21.3.1 Example #1

In this example, we'll use a number of rpmlib's header manipulation functions.

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

#include <rpm/rpmlib.h>
```

Here we've included `rpmlib.h`, which is necessary for all programs that use rpmlib.

```
void main(int argc, char ** argv)
{
  HeaderIterator iter;
    Header h, sig;
    int_32 itertag, type, count;
    void **p = NULL;
    char *blather;
    char * name;

    int fd, stat;
```

Here we've defined the program's storage. Note in particular the `HeaderIterator`, `Header`, and `int_32` datatypes.

```
    if (argc == 1) {
        fd = 0;
    } else {
        fd = open(argv[1], O_RDONLY, 0644);
    }

    if (fd < 0) {
        perror("open");
        exit(1);
    }
```

Standard stuff here. The first argument is supposed to be an RPM package file. It is opened here. If there is no argument on the command line, the program will use stdin instead.

```
    stat = rpmReadPackageInfo(fd, &sig, &h);
    if (stat) {
      fprintf(stderr,
              "rpmReadPackageInfo error status: %d\n%s\n",
              stat, strerror(errno));
        exit(stat);
    }
```

Here things start to get interesting! The signature and headers are read from package file that was just opened. If you noticed above, we've defined `sig` and `h` to be of type `Header`. That means we can use rpmlib's header-related functions on them. After a little bit of error checking, and it's time to move on...

```
    headerGetEntry(h, RPMTAG_NAME, &type, (void **) &name, &count);

    if (headerIsEntry(h, RPMTAG_PREIN)) {
        printf("There is a preinstall script for %s\n", name);
    }

    if (headerIsEntry(h, RPMTAG_POSTIN)) {
        printf("There is a postinstall script for %s\n", name);
    }
```

Now that we have the package's header, we get the package name (specified by the `RPMTAG_NAME` above). Next, we see if the package has pre-install (`RPMTAG_PREIN`) or post-install (`RPMTAG_POSTIN`) scripts. If there are, we print out a message, along with the package name.

```
    printf("Dumping signatures...\n");
    headerDump(sig, stdout, 1);

    rpmFreeSignature(sig);
```

Turning to the other `Header` structure we've read, we print out the package's signatures in human-readable form. When we're done, we free the block of signatures.

```
    printf("Iterating through the header...\n");

    iter = headerInitIterator(h);
```

Here we set up an iterator for the package's header. This will allow us to step through each entry in the header.

```
    while (headerNextIterator(iter, &itertag, &type, p, &count)) {
      switch (itertag) {
      case RPMTAG_SUMMARY:
        blather = *p;
        printf("The Summary: %s\n", blather);
        break;
      case RPMTAG_FILENAMES:
        printf("There are %d files in this package\n", count);
        break;
      }
```

This loop uses `headerNextIterator()` to return each entry's tag, type, data, and size. By using a `switch` statement on the tag, we can perform different operations on each type of entry in the header.

```
    }

    headerFreeIterator(iter);

    headerFree(h);

}
```

This is the housecleaning section of the program. First we free the iterator that we've been using, and finally the header itself. Running this program on a package gives us the following output:

```
# ./dump amanda-client-2.3.0-2.i386.rpm
There is a postinstall script for amanda-client
Dumping signatures...
Entry count: 2
Data count : 20

             CT  TAG                   TYPE           OFSET      COUNT
Entry      : 000 (1000)NAME            INT32_TYPE     0x00000000 00000001
       Data: 000 0x00029f5d (171869)
```

```
Entry      : 001 (1003)SERIAL        BIN_TYPE       0x00000004 00000016
       Data: 000 27 01 f9 97 d8 2c 36 40
       Data: 008 c6 4a 91 45 32 13 d1 62
Iterating through the header...
The Summary: Client-side Amanda package
There are 11 files in this package
#
```

21.3.2 Example #2

This example delves a bit more into the database-related side of rpmlib. After initializing rpmlib's variables by reading the appropriate `rpmrc` files, the code traverses the database records, looking for a specific package. That package's header is then dumped in its entirety.

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>

#include <rpm/rpmlib.h>
```

As before, this is the normal way of including all of rpmlib's definitions.

```
void main(int argc, char ** argv)
{
    Header h;
    int offset;
    int dspBlockNum = 0;                /* default to all */
    int blockNum = 0;
    int_32 type, count;
    char * name;
    rpmdb db;
```

Here are the data declarations. Note the declaration of `db`: this is how we will be accessing the RPM database.

```
    printf("The database path is: %s\n",
        rpmGetVar(RPMVAR_DBPATH) ? rpmGetVar(RPM_DBPATH) : "(none)");

    rpmReadConfigFiles(NULL, NULL, NULL, 0);

    printf("The database path is: %s\n",
        rpmGetVar(RPMVAR_DBPATH) ? rpmGetVar(RPM_DBPATH) : "(none)");
```

Before opening the RPM database, it's necessary to know where the database resides. This information is stored in `rpmrc` files, which are read by `rpmReadConfigFiles()`. To show that this function is really doing its job, we retrieve the RPM database path before and after the `rpmrc` files are read. Note that we test the return value of `rpmGetVar(RPM_DBPATH)` and, if it is null, we insert `(none)` in the `printf()` output. This prevents possible core dumps if no database path has been set, and besides, it's more user-friendly.

```
    if (rpmdbOpen("", &db, O_RDONLY, 0644) != 0) {
        fprintf(stderr, "cannot open /var/lib/rpm/packages.rpm\n");
        exit(1);
    }
```

Here we're opening the RPM database, and doing some cursory error checking to make sure we should continue.

```
    offset = rpmdbFirstRecNum(db);
```

We get the offset of the first database record...

```
    while (offset) {

        h = rpmdbGetRecord(db, offset);
        if (!h) {
                fprintf(stderr, "headerRead failed\n");
        exit(1);
                }
```

Here we start a `while` loop based on the record offset. As long as there is a non-zero offset (meaning that there is still an available record), we get the record. If there's a problem getting the record, we exit.

```
            headerGetEntry(h, RPMTAG_NAME, &type, (void **) &name, &count);
            if (strcmp(name, argv[1]) == 0)
              headerDump(h, stdout, 1);
```

Next, we get the package name entry from the record, and compare it with the name of the package we're interested in. If it matches, we dump the contents of the entire record.

```
        headerFree(h);

        offset = rpmdbNextRecNum(db, offset);
    }
```

At the end of the loop, we free the record, and get the offset to the next record.

```
    rpmdbClose(db);
}
```

At the end, we close the database, and exit.

Here's the program's output, edited for brevity. Notice that the database path changes from `(null)` to `/var/lib/rpm` after the `rpmrc` files are read.

```
# ./showdb amanda-client
The database path is: (null)
The database path is: /var/lib/rpm
Entry count: 37
Data count : 5219

             CT  TAG                   TYPE               OFSET      COUNT
Entry      : 000 (1000)NAME        STRING_TYPE        0x00000000 00000001
       Data: 000 amanda-client
Entry      : 001 (1001)VERSION     STRING_TYPE        0x0000000e 00000001
       Data: 000 2.3.0
Entry      : 002 (1002)RELEASE     STRING_TYPE        0x00000014 00000001
       Data: 000 7
Entry      : 003 (1004)SUMMARY     STRING_TYPE        0x00000016 00000001
       Data: 000 Client-side Amanda package
Entry      : 004 (1005)DESCRIPTION STRING_TYPE        0x00000031 00000001
...
Entry      : 017 (1027)FILENAMES   STRING_ARRAY_TYPE  0x00000df3 00000015
       Data: 000 /usr/doc/amanda-client-2.3.0-7
       Data: 001 /usr/doc/amanda-client-2.3.0-7/COPYRIGHT
       Data: 002 /usr/doc/amanda-client-2.3.0-7/INSTALL
       Data: 003 /usr/doc/amanda-client-2.3.0-7/README
       Data: 004 /usr/doc/amanda-client-2.3.0-7/SYSTEM.NOTES
       Data: 005 /usr/doc/amanda-client-2.3.0-7/WHATS.NEW
       Data: 006 /usr/doc/amanda-client-2.3.0-7/amanda-client.README
...
Entry      : 034 (1049)REQUIRENAME STRING_ARRAY_TYPE  0x0000141c 00000006
       Data: 000 libc.so.5
       Data: 001 libdb.so.2
       Data: 002 grep
       Data: 003 sed
       Data: 004 NetKit-B
       Data: 005 dump
...
#
```

As can be seen, everything that you could possibly want to know about an installed package is available using this method.

21.3.3 Example #3

This example is similar in function to the previous one, except that it uses rpmlib's search functions to find the desired package record:

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>

#include <rpm/rpmlib.h>
```

Here we include rpmlib's definitions.

```
void main(int argc, char ** argv)
{
    Header h;
    int stat;
    rpmdb db;
    dbiIndexSet matches;
```

Here are the storage declarations.

```
    if (argc != 2) {
        fprintf(stderr, "showdb2 <search term>\n");
        exit(1);
    }

    rpmReadConfigFiles(NULL, NULL, NULL, 0);

    if (rpmdbOpen("", &db, O_RDONLY, 0644) != 0) {
        fprintf(stderr, "cannot open /var/lib/rpm/packages.rpm\n");
        exit(1);
    }
```

In this section, we do some argument processing, processing the `rpmrc` files, and open the RPM database.

```
    stat = rpmdbFindPackage(db, argv[1], &matches);
    printf("Status is: %d\n", stat);
    if (stat == 0) {
      if (matches.count) {
        printf("Number of matches: %d\n", matches.count);
        h = rpmdbGetRecord(db, matches.recs[0].recOffset);
```

```
        if (h) headerDump(h, stdout, 1);
        headerFree(h);
        dbiFreeIndexRecord(matches);
      }
    }
```

In this section we use `rpmdbFindPackage()` to search for the desired package. After checking for successful status, the count of matching package records is checked. If there is at least one match, the first matching record is retrieved, and dumped. Note that there could be more than one match. Although this example doesn't dump more than the first matching record, it would be simple to access all matches by stepping through the `matches.recs` array.

Once we're done with the record, we free it, as well as the list of matching records.

```
    rpmdbClose(db);
}
```

The last thing we do before exiting is to close the database. Here's some sample output from the program. Note the successful status, and the number of matches printed before the dump:

```
# ./showdb2 rpm
Status is: 0
Number of matches: 1
Entry count: 37
Data count : 2920

             CT  TAG                   TYPE               OFSET      COUNT
Entry      : 000 (1000)NAME        STRING_TYPE        0x00000000 00000001
       Data: 000 rpm
Entry      : 001 (1001)VERSION     STRING_TYPE        0x00000004 00000001
       Data: 000 2.2.9
Entry      : 002 (1002)RELEASE     STRING_TYPE        0x0000000a 00000001
       Data: 000 1
Entry      : 003 (1004)SUMMARY     STRING_TYPE        0x0000000c 00000001
       Data: 000 Red Hat Package Manager
...
Entry      : 034 (1049)REQUIRENAME STRING_ARRAY_TYPE  0x00000b40 00000003
       Data: 000 libz.so.1
       Data: 001 libdb.so.2
       Data: 002 libc.so.5
Entry      : 035 (1050)REQUIREVERSION STRING_ARRAY_TYPE 0x00000b5f 00000003
       Data: 000
       Data: 001
       Data: 002
Entry      : 036 (1064)RPMVERSION  STRING_TYPE        0x00000b62 00000001
```

```
      Data: 000 2.2.9
#
```

Appendix A

Format of the RPM File

A.1 RPM File naming convention

While RPM will run just as well if a package file has been renamed, when the packages are created during RPM's build process, they follow a specific naming convention. The convention is:

name-version-release.architecture.rpm

where:

- *name* is a name describing the packaged software.
- *version* is the version of the packaged software.
- *release* is the number of times this version of the software has been packaged.
- *architecture* is a shorthand name describing the type of computer hardware the packaged software is meant to run on. It may also be the string `src`, or `nosrc`. Both of these strings indicate the file is an RPM source package. The `nosrc` string means that the file contains only package building files, while the `src` string means the file contains the necessary package building files *and* the software's source code.

A few notes are in order. Normally, the package name is taken verbatim from the packaged software's name. Occasionally, this approach won't work — usually this occurs when the software is split into multiple "subpackages," each supporting a different set of functions. An example of this situation would be the way `ncurses` was packaged on Red Hat Linux. The package incorporating `ncurses` basic functionality was called `ncurses`, while the package incorporating those parts of `ncurses`' program development functionality was named `ncurses-devel`.

The version number is normally taken verbatim from the package's version. The only restriction placed on the version is that it cannot contain a dash "`-`".

The release can be thought of as the *package's* version. Traditionally it is a number, starting at 1, that shows how many times the packaged software, at a given version, has been built. This is tradition and not a restriction, however. Like the version number, the only restriction is that dashes are not allowed.

The architecture specifier is a string that indicates what hardware the package has been built for. There are a number of architectures defined:

- `i386` — The Intel x86 family of microprocessors, starting with the 80386.
- `alpha` — The Digital Alpha/AXP series of microprocessors.
- `sparc` — Sun Microsystem's SPARC series of chips.
- `mips` — MIPS Technologies' processors.
- `ppc` — The Power PC microprocessor family.
- `m68k` — Motorola's 68000 series of CISC microprocessors.
- `SGI` — Equivalent to "MIPS".

This list will almost certainly change. For the most up-to-date list, please refer to the file `/usr/lib/rpmrc`. It contains information used internally by RPM, including a list of architectures and equivalent code numbers.

A.2 RPM File Format

While the following details concerning the actual format of an RPM package file were accurate at the time this was written, two points should be kept in mind:

1. The file format is subject to change.
2. If a package file is to be manipulated somehow, you are *strongly* urged to use the appropriate rpmlib routines to access the package file. Why? See point number 1!
3. This appendix describes the most recent version of the RPM file format: version 3. The `file(1)` utility can be used to see a package's file format version.

With those caveats out of the way, let's take a look inside an RPM file...

A.2.1 Parts of an RPM File

Every RPM package file can be divided into four distinct sections. They are:

- The lead.
- The signature.

- The header.
- The archive.

Package files are written to disk in network byte order. If required, RPM will automatically convert to host byte order when the package file is read. Let's take a look at each section, starting with the lead.

A.2.2 The Lead

The lead is the first part of an RPM package file. In previous versions of RPM, it was used to store information used internally by RPM. Today, however, the lead's sole purpose is to make it easy to identify an RPM package file. For example, the `file(1)` command uses the lead.[1] All the information contained in the lead has been duplicated or superseded by information contained in the header.[2]

RPM defines a C structure that describes the lead:

```
struct rpmlead {
    unsigned char magic[4];
    unsigned char major, minor;
    short type;
    short archnum;
    char name[66];
    short osnum;
    short signature_type;
    char reserved[16];
} ;
```

Let's take a look at an actual package file and examine the various pieces of data that make up the lead. In the following display, the number to the left of the colon is the byte offset, in hexadecimal, from the start of the file. The eight groups of four characters show the hex value of the bytes in the file — two bytes per group of four characters. Finally, the characters on the right show the ASCII values of the data bytes. When a data byte's value results in a non-printable character, a dot (".") is inserted instead. Here are the first thirty-two bytes of a package file — in this case, the package file `rpm-2.2.1-1.i386.rpm`:

```
00000000: edab eedb 0300 0000 0001 7270 6d2d 322e  ..........rpm-2.
00000010: 322e 312d 3100 0000 0000 0000 0000 0000  2.1-1...........
```

The first four bytes (`edab eedb`) are the magic values that identify the file as an RPM package file. Both the `file` command and RPM use these magic numbers to determine whether a file is legitimate or not.

[1] Please refer to section A.4 on page 361 for a discussion on identifying RPM package files with the `file` command.

[2] The header is discussed in section A.2.5 on page 356.

The next two bytes (0300) indicate RPM file format version. In this case, the file's major version number is 3, and the minor version number is 0. Versions of RPM later than 2.1 create version 3.0 package files.

The next two bytes (0000) determine what type of RPM file the file is. There are presently two types defined:

- Binary package file (type = 0000)
- Source package file (type = 0001)

In this case, the file is a binary package file.

The next two bytes (0001) are used to store the architecture that the package was built for. In this case, the number 1 refers to the i386 architecture.[3] In the case of a source package file, these two bytes should be ignored, as source packages are not built for a specific architecture.

The next sixty-six bytes (starting with `7270 6d2d`) contain the name of the package. The name must end with a null byte, which leaves sixty-five bytes for RPM's usual `name-version-release`-style name. In this case, we can read the name from the right side of the output:

```
rpm-2.2.1-1
```

Since the name `rpm-2.2.1-1` is shorter than the sixty-five bytes allocated for the name, the leftover bytes are filled with nulls.

Skipping past the space allocated for the name, we see two bytes (0001):

```
00000040: 0000 0000 0000 0000 0000 0000 0001 0005  ................
00000050: 0400 0000 24e1 ffbf 6bb3 0008 00e6 ffbf  ....$...k.......
```

These bytes represent the operating system for which this package was built. In this case, 1 equals Linux. As with the architecture-to-number translations, the operating system and corresponding code numbers can be found in the file, `/usr/lib/rpmrc`.

The next two bytes (0005) indicate the type of signature used in the file. A type 5 signature is new to version 3 RPM files. The signature appears next in the file, but we need to discuss an additional detail before exploring the signature.

A.2.3 Wanted: A New RPM Data Structure

By looking at the C structure that defines the lead, and matching it with the bytes in an actual package file, it's trivial to extract the data from the lead. From a programming standpoint, it's also easy to manipulate data in the lead — It's simply a matter of using the element names from the structure. But there's a problem. And because of that problem the lead is no longer used internally by RPM.

[3] It should be noted that the architecture used internally by RPM is actually stored in the header. This value is strictly for `file(1)`'s use.

The lead: An Abandoned Data Structure

What's the problem, and why is the lead no longer used by RPM? The answer to these questions is a single word: inflexibility. The technique of defining a C structure to access data in a file just isn't very flexible. Let's look at an example.

Flip back to the lead's C structure on page 349. Say, for example, that some software comes along, and it has a long name. A *very* long name. A name so long, in fact, that the 66 bytes defined in the structure element `name` just couldn't hold it.

What can we do? Well, we could certainly change the structure such that the `name` element would be 100 bytes long. But once a new version of RPM is created using this new structure, we have two problems:

1. Any package file created with the new version of RPM wouldn't be able to read older package formats.
2. Any older version of RPM would be unable to install packages created with the newer version of RPM.

Not a very good situation! Ideally, we would like to somehow eliminate the requirement that the format of the data written to a package file be engraved in granite. We should be able to do the following things, all without losing compatibility with existing versions of RPM.

- Add extra data to the file format.
- Change the size of existing data.
- Reorder the data.

Sounds like a big problem, but there's a solution...

Is There a Solution?

The solution is to standardize the method by which information is retrieved from a file. This is done by creating a well-defined data structure that contains easily searched information about the data, and then physically separating that information from the data.

When the data is required, it is found by using the easily searched information, which points to the data itself. The benefits are, that the data can be placed anywhere in the file, and that the format of the data itself can change.

The Solution: The Header Structure

The header structure is RPM's solution to the problem of easily manipulating information in a standard way. The header structure's sole purpose in life is to contain zero or more pieces of data. A file can have more than one header structure in it. In fact, an RPM package file has two — the signature, and the header. It was from this header that the header structure got its name.

There are three sections to each header structure. The first section is known as the "header structure header". The header structure header is used to identify the start of a header structure, its size, and the number of data items it contains.

Following the header structure header is an area called the "index". The index contains one or more index entries. Each index entry contains information about, and a pointer to, a specific data item.

After the index comes the "store". It is in the store that the data items are kept. The data in the store is packed together as closely as possible. The order in which the data is stored is immaterial — a far cry from the C structure used in the lead.

The Header Structure in Depth

Let's take a more in-depth look at the actual format of a header structure, starting with the header structure header:

The Header Structure Header — The header structure header always starts with a three-byte magic number: `8e ad e8`. Following this is a one-byte version number. Next are four bytes that are reserved for future expansion. After the reserved bytes, there is a four-byte number that indicates how many index entries exist in this header structure, followed by another four-byte number indicating how many bytes of data are part of the header structure.

The Index Entry — The header structure's index is made up of zero or more index entries. Each entry is sixteen bytes longs. The first four bytes contain a "tag" — a numeric value that identifies what type of data is pointed to by the entry. The tag values change according to the header structure's position in the RPM file. A list of the actual tag values, and what they represent, will be included later in this appendix.

Following the tag, is a four-byte "type", which is a numeric value that describes the format of the data pointed to by the entry. The types and their values do not change from header structure to header structure. Here is the current list:

- NULL = 0
- CHAR = 1
- INT8 = 2
- INT16 = 3
- INT32 = 4
- INT64 = 5
- STRING = 6
- BIN = 7

- STRING_ARRAY = 8

A few of the data types might need some clarification. The STRING data type is simply a null-terminated string, while the STRING_ARRAY is a collection of strings. Finally, the BIN data type is a collection of binary data. This is normally used to identify data that is longer than an INT, but not a printable STRING.

Next is a four-byte "offset" that contains the position of the data, relative to the beginning of the store. We'll talk about the store in just a moment.

Finally, there is a four-byte "count" that contains the number of data items pointed to by the index entry. There are a few wrinkles to the meaning of the count, and they center around the STRING and STRING_ARRAY data types. STRING data always has a count of 1, while STRING_ARRAY data has a count equal to the number of strings contained in the store.

The Store — The store is where the data contained in the header structure is stored. Depending on the data type being stored, there are some details that should be kept in mind:

- For STRING data, each string is terminated with a null byte.
- For INT data, each integer is stored at the natural boundary for its type. A 64-bit INT is stored on an 8-byte boundary, a 16-bit INT is stored on a 2-byte boundary, and so on.
- All data is in network byte order.

With all these details out of the way, let's take a look at the signature.

A.2.4 The Signature

The signature section follows the lead in the RPM package file. It contains information that can be used to verify the integrity, and optionally, the authenticity of the majority of the package file. The signature is implemented as a header structure.

You probably noticed the word, "majority", above. The information in the signature header structure is based on the contents of the package file's header and archive only. The data in the lead and the signature header structure are not included when the signature information is created, nor are they part of any subsequent checks based on that information.

While that omission might seem to be a weakness in RPM's design, it really isn't. In the case of the lead, since it is used only for easy identification of package files, any changes made to that part of the file would, at worst, leave the file in such a state that RPM wouldn't recognize it as a valid package file. Likewise, any changes to the signature header structure would make it impossible to verify the file's integrity, since the signature information would have been changed from their original values.

Analyzing the Signature Area

Using our new-found knowledge of header structures, let's take a look at the signatures in `rpm-2.2.1-1.i386.rpm`:

```
00000060: 8ead e801 0000 0000 0000 0003 0000 00ac  ................
```

The first three bytes (`8ead e8`) contain the magic number for the start of the header structure. The next byte (`01`) is the header structure's version.

As we discussed earlier, the next four bytes (`0000 0000`) are reserved. The four bytes after that (`0000 0003`) represent the number of index entries in the signature section, namely, three. Following that are four bytes (`0000 00ac`) that indicate how many bytes of data are stored in the signature. The hex value `00ac`, when converted to decimal, means the store is 172 bytes long.

Following the first 16 bytes is the index. Each of the three index entries in this header structure consists of four 32-bit integers, in the following order:

- Tag
- Type
- Offset
- Count

Let's take a look at the first index entry:

```
00000070: 0000 03e8 0000 0004 0000 0000 0000 0001  ................
```

The tag consists of the first four bytes (`0000 03e8`), which is 1000 when translated from hex. Looking in the RPM source directory at the file `lib/signature.h`, we find the following tag definitions:

```
#define SIGTAG_SIZE          1000
#define SIGTAG_MD5           1001
#define SIGTAG_PGP           1002
```

So the tag we are studying is for a size signature. Let's continue.

The next four bytes (`0000 0004`) contain the data type. As we saw earlier, data type 4 means that the data stored for this index entry, is a 32-bit integer. Skipping the next four bytes for a moment, the last four bytes (`0000 0001`) are the number of 32-bit integers pointed to by this index entry.

Now, let's go back to the four bytes prior to the count (`0000 0000`). This number is the offset, in bytes, at which the size signature is located. It has a value of zero, but the question is, zero bytes from what? The answer, although it doesn't do us much good, is that the offset is calculated from the start of the store. So first we must find where the store begins, and we can do that by performing a simple calculation.

First, go back to the start of the signature section. (We've made a copy here so you won't need to flip from page to page)

```
00000060: 8ead e801 0000 0000 0000 0003 0000 00ac  ................
```

After the magic, the version, and the four reserved bytes, there is the number of index entries (`0000 0003`). Since we know that each index entry is sixteen bytes long (four for the tag, four for the type, four for the offset, and four for the count), we can multiply the number of entries (3) by the number of bytes in each entry (16), and obtain the total size of the index, which is 48 decimal, or 30 in hex. Since the first index entry starts at hex offset 70, we can simply add hex 30 to hex 70, and get, in hex, offset a0. So let's skip down to offset a0, and see what's there:

```
000000a0: 0004 4c4f b025 b097 1597 0132 df35 d169  ..LO.%.....2.5.i
```

If we've done our math correctly, the first four bytes (`0004 4c4f`) should represent the size of this file. Converting to decimal, this is 281,679. Let's take a look at the size of the actual file:

```
# ls -al rpm-2.2.1-1.i386.rpm
-rw-rw-r-- 1 ed ed 282015 Jul 21 16:05 rpm-2.2.1-1.i386.rpm
#
```

Hmmm, something's not right. Or is it? It looks like we're short by 336 bytes, or in hex, 150. Interesting how that's a nice round hex number, isn't it? For now, let's continue through the remainder of the index entries, and see if hex 150 pops up elsewhere.

Here's the next index entry. It has a tag of decimal 1001, which is an MD5 checksum. It is type 7, which is the BIN data type, it is 16 bytes long, and its data starts four bytes after the beginning of the store:

```
00000080: 0000 03e9 0000 0007 0000 0004 0000 0010  ................
```

And here's the data. It starts with `b025` (Remember that offset of four!) and ends on the second line with `5375`. This is a 128-bit MD5 checksum of the package file's header and archive sections.

```
000000a0: 0004 4c4f b025 b097 1597 0132 df35 d169  ..LO.%.....2.5.i
000000b0: 329c 5375 8900 9503 0500 31ed 6390 a520  2.Su......1.c..
```

Ok, let's jump back to the last index entry:

```
00000090: 0000 03ea 0000 0007 0000 0014 0000 0098  ................
```

It has a tag value of `03ea` (1002 in decimal — a PGP signature block) and is also a BIN data type. The data starts 20 decimal bytes from the start of the data area, which would put it at file offset b4 (in hex). It's a biggie — 152 bytes long! Here's the data, starting with `8900`:

```
000000b0: 329c 5375 8900 9503 0500 31ed 6390 a520  2.Su......1.c..
000000c0: e8f1 cba2 9bf9 0101 437b 0400 9c8e 0ad4  ........C{......
000000d0: 3790 364e dfb0 9a8a 22b5 b0b3 dc30 4c6f  7.6N...."....0Lo
000000e0: 91b8 c150 704e 2c64 d88a 8fca 18ab 5b6f  ...PpN,d......[o
000000f0: f041 ebc8 d18a 01c9 3601 66f0 9ddd e956  .A......6.f....V
00000100: 3142 61b3 b1da 8494 6bef 9c19 4574 c49f  1Ba.....k...Et..
00000110: ee17 35e1 d105 fb68 0ce6 715a 60f1 c660  ..5....h..qZ`..`
00000120: 279f 0306 28ed 0ba0 0855 9e82 2b1c 2ede  '...(....U..+...
00000130: e8e3 5090 6260 0b3c ba04 69a9 2573 1bbb  ..P.b`.<..i.%s..
00000140: 5b65 4de1 b1d2 c07f 8afa 4a9b 0000 0000  [eM.......J.....
```

It ends with the bytes `4a9b`. This is a 1,216-bit PGP signature block. It is also the end of the signature section. There are four null bytes following the last data item in order to round the size out so that it ends on an 8-byte boundary. This means that the offset of the next section starts at offset 150, in hex. Say, wasn't the size in the size signature off by 150 hex? Yes, the size in the signature is the size of the file — *minus* the size of the lead and the signature sections.

A.2.5 The Header

The header section contains all available information about the package. Entries such as the package's name, version, and file list, are contained in the header. Like the signature section, the header is in header structure format. Unlike the signature, which has only three possible tag types, the header has more than *sixty* different tags. The list of currently defined tags appears later in this appendix on page 357. Be aware that the list of tags changes frequently — the definitive list appears in the RPM sources in `lib/rpmlib.h`.

Analyzing the Header

The easiest way to find the start of the header is to look for the second header structure by scanning for its magic number (`8ead e8`). The sixteen bytes, starting with the magic, are the header structures's header. They follow the same format as the header in the signature's header structure:

```
00000150: 8ead e801 0000 0000 0000 0021 0000 09d3  ...........!....
```

As before, the byte following the magic identifies this header structure as being in version 1 format. Following the four reserved bytes, we find the count of entries stored in the header (`0000 0021`). Converting to decimal, we find that there are 33 entries in the header. The next four bytes (`0000 09d3`) converted to decimal, tell us that there are 2,515 bytes of data in the store.

Since the header is a header structure just like the signature, we know that the next 16 bytes are the first index entry:

```
00000160: 0000 03e8 0000 0006 0000 0000 0000 0001  ................
```

The first four bytes (`0000 03e8`) are the tag, which is the tag for the package name. The next four bytes indicate the data is type 6, or a null-terminated string. There's an offset of zero in the next four bytes, meaning that the data for this tag is first in the store. Finally, the last four bytes (`0000 0001`) show that the data count is 1, which is the only legal value for data of type STRING.

To find the data, we need to take the offset from the start of the first index entry in the header (160), and add in the count of index entries (21) multiplied by the size of an index entry (10). Doing the math (all the values shown, are in hex, remember!), we arrive at the offset to the store, hex 370. Since the offset for this particular index entry is zero, the data should start at offset 370:

```
00000370: 7270 6d00 322e 322e 3100 3100 5265 6420  rpm.2.2.1.1.Red
```

Since the data type for this entry is a null-terminated string, we need to keep reading bytes until we reach a byte whose numeric value is zero. We find the bytes `72`, `70`, `6d`, and `00` — a null. Looking at the ASCII display on the right, we find that the bytes form the string `rpm`, which is the name of this package.

Now for a slightly more complicated example. Let's look at the following index entry:

```
00000250: 0000 0403 0000 0008 0000 0199 0000 0018  ................
```

Tag 403 means that this entry is a list of filenames. The data type 8, or STRING_ARRAY, seems to bear this out. From the previous example, we found that the data area for the header began at offset 370. Adding the offset to the first filename (199), gives us 509. Finally, the count of 18 hex means that there should be 24 null-terminated strings containing filenames:

```
00000500: 696e 6974 6462 0a0a 002f 6269 6e2f 7270  initdb.../bin/rp
00000510: 6d00 2f65 7463 2f72 706d 7263 002f 7573  m./etc/rpmrc./us
```

The byte at offset 509 is 2f — a "/". Reading up to the first null byte, we find that the first filename is `/bin/rpm`, followed by `/etc/rpmrc`. This continues on for 22 more filenames.

There are many more tags that we could decode, but they are all done in the same manner.

Header Tag Listing

The following list shows the tags available, along with their defined values, for use in the header. This list is current as of version 2.3 of RPM. For the most up-to-date version, look in the file `lib/rpmlib.h` in the latest version of the RPM sources.

```
#define RPMTAG_NAME                     1000
#define RPMTAG_VERSION                  1001
#define RPMTAG_RELEASE                  1002
#define RPMTAG_SERIAL                   1003
#define RPMTAG_SUMMARY                  1004
#define RPMTAG_DESCRIPTION              1005
#define RPMTAG_BUILDTIME                1006
#define RPMTAG_BUILDHOST                1007
#define RPMTAG_INSTALLTIME              1008
#define RPMTAG_SIZE                     1009
#define RPMTAG_DISTRIBUTION             1010
#define RPMTAG_VENDOR                   1011
#define RPMTAG_GIF                      1012
#define RPMTAG_XPM                      1013
#define RPMTAG_COPYRIGHT                1014
#define RPMTAG_PACKAGER                 1015
#define RPMTAG_GROUP                    1016
#define RPMTAG_CHANGELOG                1017
#define RPMTAG_SOURCE                   1018
#define RPMTAG_PATCH                    1019
#define RPMTAG_URL                      1020
#define RPMTAG_OS                       1021
#define RPMTAG_ARCH                     1022
#define RPMTAG_PREIN                    1023
#define RPMTAG_POSTIN                   1024
#define RPMTAG_PREUN                    1025
#define RPMTAG_POSTUN                   1026
#define RPMTAG_FILENAMES                1027
#define RPMTAG_FILESIZES                1028
#define RPMTAG_FILESTATES               1029
#define RPMTAG_FILEMODES                1030
#define RPMTAG_FILEUIDS                 1031
#define RPMTAG_FILEGIDS                 1032
#define RPMTAG_FILERDEVS                1033
#define RPMTAG_FILEMTIMES               1034
#define RPMTAG_FILEMD5S                 1035
#define RPMTAG_FILELINKTOS              1036
#define RPMTAG_FILEFLAGS                1037
#define RPMTAG_ROOT                     1038
#define RPMTAG_FILEUSERNAME             1039
#define RPMTAG_FILEGROUPNAME            1040
#define RPMTAG_EXCLUDE                  1041 /* not used */
#define RPMTAG_EXCLUSIVE                1042 /* not used */
#define RPMTAG_ICON                     1043
#define RPMTAG_SOURCERPM                1044
#define RPMTAG_FILEVERIFYFLAGS          1045
```

```
#define RPMTAG_ARCHIVESIZE             1046
#define RPMTAG_PROVIDES                1047
#define RPMTAG_REQUIREFLAGS            1048
#define RPMTAG_REQUIRENAME             1049
#define RPMTAG_REQUIREVERSION          1050
#define RPMTAG_NOSOURCE                1051
#define RPMTAG_NOPATCH                 1052
#define RPMTAG_CONFLICTFLAGS           1053
#define RPMTAG_CONFLICTNAME            1054
#define RPMTAG_CONFLICTVERSION         1055
#define RPMTAG_DEFAULTPREFIX           1056
#define RPMTAG_BUILDROOT               1057
#define RPMTAG_INSTALLPREFIX           1058
#define RPMTAG_EXCLUDEARCH             1059
#define RPMTAG_EXCLUDEOS               1060
#define RPMTAG_EXCLUSIVEARCH           1061
#define RPMTAG_EXCLUSIVEOS             1062
#define RPMTAG_AUTOREQPROV             1063 /* used internally by build */
#define RPMTAG_RPMVERSION              1064
#define RPMTAG_TRIGGERSCRIPTS          1065
#define RPMTAG_TRIGGERNAME             1066
#define RPMTAG_TRIGGERVERSION          1067
#define RPMTAG_TRIGGERFLAGS            1068
#define RPMTAG_TRIGGERINDEX            1069
#define RPMTAG_VERIFYSCRIPT            1079
```

A.2.6 The Archive

Following the header section is the archive. The archive holds the actual files that comprise the package. The archive is compressed using GNU zip. We can verify this if we look at the start of the archive:

```
00000d40: 0000 001f 8b08 0000 0000 0002 03ec fd7b  ...............{
00000d50: 7c13 d516 388e 4e92 691b 4a20 010a 1428  |...8.N.i.J ...(
```

In this example, the archive starts at offset `d43`. According to the contents of `/usr/lib/magic`, the first two bytes of a `gzipped` file should be `1f8b`, which is, in fact, what we see. The following byte (`08`) is the flag used by GNU zip to indicate the file has been compressed with `gzip`'s "deflation" method. The eighth byte has a value of `02`, which means that the archive has been compresed using `gzip`'s maximum compression setting. The following byte contains a code indicating the operating system under which the archive was compressed. A `03` in this byte indicates that the compression ran under a UNIX-like operating system.

The remainder of the RPM package file is the compressed archive. After the archive is uncompressed, it is an ordinary `cpio` archive in SVR4 format with a CRC checksum.

A.3 Tools For Studying RPM Files

In the `tools` directory packaged with the RPM sources, are a number of small programs that use the RPM library to extract the various sections of a package file. Normally used by the RPM developers for debugging purposes, these tools can also be used to make it easier to understand the RPM package file format. Here is a list of the programs, and what they do:

- `rpmlead` — Extracts the lead section from a package file.
- `rpmsignature` — Extracts the signature section from a package file.
- `rpmheader` — Extracts the header from a package file.
- `rpmarchive` — Extracts the archive from a package file.
- `dump` — Displays a header structure in an easily readable format.

The first four programs take an RPM package file as their input. The package file can be read either from standard input, or by including the file name on the command line. In either case, the programs write to standard output. Here is how `rpmlead` can be used to display the lead from a package file:

```
# rpmlead foo.rpm | od -x
0000000 abed dbee 0003 0000 0100 7072 2d6d 2e32
0000020 2e32 2d31 0031 0000 0000 0000 0000 0000
0000040 0000 0000 0000 0000 0000 0000 0000 0000

0000100 0000 0000 0000 0000 0000 0000 0100 0500
0000120 0004 0000 e124 bfff b36b 0800 e600 bfff
0000140
#
```

Since each of these programs can also act as filters, the following command is equivalent to the one above:

```
# cat foo.rpm | rpmlead | od -x
0000000 abed dbee 0003 0000 0100 7072 2d6d 2e32
0000020 2e32 2d31 0031 0000 0000 0000 0000 0000
0000040 0000 0000 0000 0000 0000 0000 0000 0000

0000100 0000 0000 0000 0000 0000 0000 0100 0500
0000120 0004 0000 e124 bfff b36b 0800 e600 bfff
0000140
#
```

The `dump` program is used in conjunction with `rpmsignature` or `rpmheader`. It makes decoding header structures a snap:

```
# rpmsignature foo.rpm | dump
Entry count:  3
Data count :  172


             CT  TAG                   TYPE             OFSET      COUNT
Entry      : 000 (1000)NAME            INT32_TYPE       0x00000000 00000001
       Data: 000 0x00044c4f (281679)
Entry      : 001 (1001)VERSION         BIN_TYPE         0x00000004 00000016
       Data: 000 b0 25 b0 97 15 97 01 32
       Data: 008 df 35 d1 69 32 9c 53 75
Entry      : 002 (1002)RELEASE         BIN_TYPE         0x00000014 00000152
       Data: 000 89 00 95 03 05 00 31 ed
       Data: 008 63 90 a5 20 e8 f1 cb a2
       Data: 016 9b f9 01 01 43 7b 04 00
       Data: 024 9c 8e 0a d4 37 90 36 4e
       Data: 032 df b0 9a 8a 22 b5 b0 b3
       Data: 040 dc 30 4c 6f 91 b8 c1 50
       Data: 048 70 4e 2c 64 d8 8a 8f ca
       Data: 056 18 ab 5b 6f f0 41 eb c8
       Data: 064 d1 8a 01 c9 36 01 66 f0
       Data: 072 9d dd e9 56 31 42 61 b3
       Data: 080 b1 da 84 94 6b ef 9c 19
       Data: 088 45 74 c4 9f ee 17 35 e1
       Data: 096 d1 05 fb 68 0c e6 71 5a
       Data: 104 60 f1 c6 60 27 9f 03 06
       Data: 112 28 ed 0b a0 08 55 9e 82
       Data: 120 2b 1c 2e de e8 e3 50 90
       Data: 128 62 60 0b 3c ba 04 69 a9
       Data: 136 25 73 1b bb 5b 65 4d e1
       Data: 144 b1 d2 c0 7f 8a fa 4a 9b

 #
```

One aspect of `dump` worth noting, is that it is optimized for decoding the header section of a package file. When used with `rpmsignature`, it displays the tag names used in the header, instead of the signature tag names. The data is displayed properly in either case, however.

A.4 Identifying RPM files with the `file(1)` command

The `magic` file on most UNIX-like systems today should have the necessary information to identify RPM files. But in case your system doesn't, the following information can be added to the file:

```
#-------------------------------------------------------------------
#
# RPM: file(1) magic for Red Hat Packages
#
0       beshort         0xedab
>2      beshort         0xeedb          RPM
>>4     byte            x               v%d
>>6     beshort         0               bin
>>6     beshort         1               src
>>8     beshort         1               i386
>>8     beshort         2               Alpha
>>8     beshort         3               Sparc
>>8     beshort         4               MIPS
>>8     beshort         5               PowerPC
>>8     beshort         6               68000
>>8     beshort         7               SGI
>>10    string          x               %s
```

The output of the `file` command is succinct:

```
# file baz
baz:  RPM v3 bin i386 vlock-1.0-2
#
```

In this case, the file called `baz` is a version 3 format RPM file containing release 2 of version 1.0 of the `vlock` package, which has been built for the Intel x86 architecture.

Appendix B

The `rpmrc` File

The `rpmrc` file is used to control RPM's actions. The file's entries have an effect on nearly every aspect of RPM's operations. Here, we describe the `rpmrc` files in more detail, as well as the command used to show how RPM interprets the files.

B.1 Using the `--showrc` Option

As we'll see in a moment, RPM can read more than one `rpmrc` file, and each file can contain nearly thirty different types of entries. This can make it difficult to determine what values RPM is actually using.

Luckily, there's an option that can be used to help make sense of it all. The `--showrc` option displays the value for each of the entries. The output is divided into two sections:

1. Architecture and operating system values.
2. `rpmrc` values.

The architecture and operating system values define the architecture and operating system that RPM is running on. These values define the environment for both building and installing packages. They also define which architectures and operating systems are compatible with each other.

The `rpmrc` values define many aspects of RPM's operation. These values range from the path to RPM's database, to the name of the person listed as having built the package.

Here's an example of `--showrc`'s output:

```
# rpm --showrc
ARCHITECTURE AND OS:
build arch          : i386
build os            : Linux
```

```
install arch         : i486
install os           : Linux
compatible arch list : i486 i386
compatible os list   : Linux
RPMRC VALUES:
builddir             : /usr/src/redhat/BUILD
buildroot            : (not set)
cpiobin              : cpio
dbpath               : /var/lib/rpm
defaultdocdir        : /usr/doc
distribution         : (not set)
excludedocs          : (not set)
ftpport              : (not set)
ftpproxy             : (not set)
messagelevel         : (not set)
netsharedpath        : (not set)
optflags             : -O2 -m486 -fno-strength-reduce
packager             : (not set)
pgp_name             : (not set)
pgp_path             : (not set)
require_distribution : (not set)
require_icon         : (not set)
require_vendor       : (not set)
root                 : (not set)
rpmdir               : /usr/src/redhat/RPMS
signature            : none
sourcedir            : /usr/src/redhat/SOURCES
specdir              : /usr/src/redhat/SPECS
srcrpmdir            : /usr/src/redhat/SRPMS
timecheck            : (not set)
tmppath              : /var/tmp
topdir               : /usr/src/redhat
vendor               : (not set)
#
```

As you can see, the `--showrc` option clearly displays the values RPM will use. `--showrc` can also be used with the `--rcfile` option, which makes it easy to see the effect of specifying a different `rpmrc` file.

B.2 Different Places an rpmrc File Resides

RPM looks for `rpmrc` files in four places:

1. In `/usr/lib/`, for a file called `rpmrc`.
2. In `/etc/`, for a file called `rpmrc`.

3. In a file called `.rpmrc` in the user's login directory.
4. In a file specified by the `--rcfile` option, if the option is present on the command line.

The first three files are read in the order listed, such that if a given `rmprc` entry is present in each file, the value of the entry read *last* is the one used by RPM. This means, for example, that an entry in `.rpmrc` in the user's login directory will always override the same entry in `/etc/rpmrc`. Likewise, an entry in `/etc/rpmrc` will always override the same entry in `/usr/lib/rpmrc`.

If the `--rcfile` option is used, then only `/usr/lib/rpmrc` and the file following the `--rcfile` option are read, in that order. The `/usr/lib/rpmrc` file is *always* read first. This cannot be changed.

Let's look at each of these files, starting with `/usr/lib/rpmrc`.

B.2.1 /usr/lib/rpmrc

The file `/usr/lib/rpmrc` is *always* read. It contains information that RPM uses to set some default values. *This file should never be modified!* Doing so may cause RPM to operate incorrectly.

After this stern warning, we should note that it's perfectly all right to look at it. Here it is, in fact:

```
#############################################################
# Default values, often overridden in /etc/rpmrc

dbpath: /var/lib/rpm
topdir: /usr/src/redhat
tmppath: /var/tmp
cpiobin: cpio
defaultdocdir:  /usr/doc

#############################################################

# Please send new entries to rpm-list@redhat.com

#############################################################
# Values for RPM_OPT_FLAGS for various platforms

optflags: i386 -O2 -m486 -fno-strength-reduce
optflags: alpha -O2
optflags: sparc -O2
optflags: m68k -O2 -fomit-frame-pointer

#############################################################
```

```
# Canonical arch names and numbers

arch_canon: i986: i986 1
arch_canon: i886: i886 1
arch_canon: i786: i786 1
arch_canon: i686: i686 1
arch_canon: i586: i586 1
arch_canon: i486: i486 1
arch_canon: i386: i386 1
arch_canon: alpha: alpha 2
arch_canon:  sparc: sparc 3
arch_canon:  sun4: sparc 3
arch_canon:  sun4m: sparc 3
arch_canon:  sun4c: sparc 3
# This is really a place holder for MIPS.
arch_canon: mips: mips 4
arch_canon: ppc: ppc 5
# This is really a place holder for 68000
arch_canon: m68k: m68k 6
# This is wrong. We really need globbing in here :-(
arch_canon: IP: sgi 7
arch_canon:     IP22:   sgi     7

arch_canon:    9000/712:       hppa1.1 9

arch_canon:    sun4u:   usparc  10

#############################################################
# Canonical OS names and numbers

os_canon: Linux: Linux 1
os_canon: IRIX: Irix 2
# This is wrong
os_canon: SunOS5: solaris 3
os_canon: SunOS4: SunOS 4

os_canon:      AmigaOS: AmigaOS 5
os_canon:          AIX: AIX     5
os_canon:        HP-UX: hpux10  6
os_canon:         OSF1: osf1    7
os_canon:      FreeBSD: FreeBSD 8

#############################################################
# For a given uname().machine, the default build arch

buildarchtranslate: osfmach3_i986: i386
```

```
buildarchtranslate: osfmach3_i886: i386
buildarchtranslate: osfmach3_i786: i386
buildarchtranslate: osfmach3_i686: i386
buildarchtranslate: osfmach3_i586: i386
buildarchtranslate: osfmach3_i486: i386
buildarchtranslate: osfmach3_i386: i386

buildarchtranslate: i986: i386
buildarchtranslate: i886: i386
buildarchtranslate: i786: i386
buildarchtranslate: i686: i386
buildarchtranslate: i586: i386
buildarchtranslate: i486: i386
buildarchtranslate: i386: i386

buildarchtranslate: osfmach3_ppc: ppc

#############################################################
# Architecture compatibility

arch_compat: alpha: axp

arch_compat: i986: i886
arch_compat: i886: i786
arch_compat: i786: i686
arch_compat: i686: i586
arch_compat: i586: i486
arch_compat: i486: i386

arch_compat: osfmach3_i986: i986 osfmach3_i886
arch_compat: osfmach3_i886: i886 osfmach3_i786
arch_compat: osfmach3_i786: i786 osfmach3_i686
arch_compat: osfmach3_i686: i686 osfmach3_i586
arch_compat: osfmach3_i586: i586 osfmach3_i486
arch_compat: osfmach3_i486: i486 osfmach3_i386
arch_compat: osfmach3_i386: i486

arch_compat: osfmach3_ppc: ppc

arch_compat: usparc: sparc
```

Quite a bunch of stuff, isn't it? With the exception of the first five lines, which indicate where several important directories and programs are located, the remainder of this file contains `rpmrc` entries that are related to RPM's architecture and operating system processing. As you might imagine, any tinkering here will probably not be very productive, so leave any modifications here to the RPM developers.

Next, we have /etc/rpmrc.

B.2.2 /etc/rpmrc

The file /etc/rpmrc, unlike /usr/lib/rpmrc, is fair game for modifications and additions. In fact, /etc/rpmrc isn't created by default, so its contents are entirely up to you. It's the perfect place to keep rpmrc entries of a system-wide or global nature.

The vendor entry is a great example of a good candidate for inclusion in /etc/rpmrc. In most cases, a particular system is dedicated to building packages for one vendor. In these instances, setting the vendor entry in /etc/rpmrc is best.

Next in the hierarchy is a file named .rpmrc, residing in the user's login directory.

B.2.3 .rpmrc in the user's login directory

As you might imagine, a file called .rpmrc in a user's login directory is only going to be read by that user when he or she runs RPM. Like /etc/rpmrc, this file is not created by default, but it can contain the same rpmrc entries as the other files. The packager entry, which should contain the name and contact information for the person who built the package, is an appropriate candidate for ~/.rpmrc.

B.2.4 File indicated by the --rcfile option

The --rcfile option is best used only when a totally different RPM configuration is desired for one or two packages. Since the only other rpmrc file read is /usr/lib/rpmrc with its low-level default settings, the file specified with the --rcfile option will have to be more comprehensive than either /etc/rpmrc or ~/.rmprc.

B.3 rpmrc File Syntax

As you might have surmised from the example file we briefly reviewed, the basic syntax of an rpmrc file entry is:

<name>:<value>

The <name> part of the entry is not case sensitive, so any capitalization is acceptable. The colon separating the name from its value must immediately follow the name. No spaces are allowed here. The formatting requirements on the value side of the entry vary from value to value and will be discussed along with each entry.

B.4 rpmrc File Entries

In this section, we discuss the various entries that can be used in each of the rpmrc files.

B.4.1 arch_canon

The arch_canon entry is used to define a table of architecture names and their associated numbers. These canonical architecture names and numbers are then used internally by RPM whenever architecture-specific processing takes place. This entry's format is:

```
arch_canon:<label>: <string> <value>
```

The <label> is compared against information from uname(2) after it's been translated using the appropriate buildarchtranslate entry. If a match is found, then <string> is used by RPM to reference the system's architecture. When building a binary package, RPM uses <string> as part of the package's filename, for instance.

The <value> is a numeric value RPM uses internally to identify the architecture. For example, this number is written in the header of each package file so that the file command can identify the architecture for which the package was built.

B.4.2 os_canon

The os_canon entry is used to define a table of operating system names and their associated numbers. These canonical operating system names and numbers are then used internally by RPM whenever operating system-specific processing takes place. This entry's format is:

```
os_canon:<label>: <string> <value>
```

The <label> is compared against information from uname(2) after it's been translated using the appropriate buildostranslate entry.[1] If a match is found, then <string> is used by RPM to reference the operating system.

The <value> is a numeric value used to uniquely identify the operating system.

B.4.3 buildarchtranslate

The buildarchtranslate entry is used in the process of defining the architecture that RPM will use as the "build" architecture. As the name implies, it is used to translate the raw information returned from uname(2) to the canonical architecture defined by arch_canon.

The format of the buildarchtranslate entry is slightly different from most other rpmrc file entries. Instead of the usual <name>:<value> format, the buildarchtranslate entry looks like this:

```
buildarchtranslate:<label>: <string>
```

The <label> is compared against information from uname(2). If a match is found, then <string> is used by RPM to define the build architecture.

[1]The buildostranslate rpmrc file entry is discussed on page 370.

B.4.4 buildostranslate

The buildostranslate entry is used in the process of defining the operating system RPM will use as the "build" operating system. As the name implies, it is used to translate the raw information returned by uname(2) to the canonical operating sysetm defined by os_canon.

The format of the buildostranslate entry is slightly different from most other rpmrc file entries. Instead of the usual <name>:<value> format, the buildostranslate entry looks like this:

```
buildostranslate:<label>:  <string>
```

The <label> is compared against information from uname(2). If a match is found, then <string> is used by RPM to define the build operating system.

B.4.5 arch_compat

The arch_compat entry is used to define which architectures are compatible with one another. This information is used when packages are installed; in this way, RPM can determine whether a given package file is compatible with the system. The format of the entry is:

```
arch_compat:<label>:  <list>
```

The <label> is an architecture string, as defined by an arch_canon entry. The <list> following it consists of one or more architectures, also defined by arch_canon. If there is more than one architecture in the list, they should be separated by a space.

The architectures in the list are considered compatible to the architecture specified in the label.

B.4.6 os_compat

Default value: (operating system-specific)

The os_compat entry is used to define which operating systems are compatible with one another. This information is used when packages are installed; in this way, RPM can determine whether a given package file is compatible with the system. The format of the entry is:

```
<name>:<label>:  <list>
```

The <label> is an operating system string, as defined by an os_canon entry. The <list> following it consists of one or more operating systems, also defined by os_canon. If there is more than one operating system in the list, they should be separated by a space.

The operating systems in the list are considered compatible to the operating system specified in the label.

B.4.7 builddir

Default value: (topdir)/BUILD

The builddir entry is used to define the path to the directory in which RPM will build packages. Its default value is taken from the value of the topdir entry, with "/BUILD" appended to it. Note that if you redefine builddir, you'll need to specify a complete path.

B.4.8 buildroot

Default value: (none)

The buildroot entry defines the path used as the root directory during the install phase of a package build. For more information on using build roots, please see chapter 16, specifically, section 16.1 on page 230.

B.4.9 cpiobin

Default value: cpio

The cpiobin entry is used to define the name (and optionally, path) of the cpio program. RPM uses cpio to perform a variety of functions, and needs to know where the program can be found.

B.4.10 dbpath

Default value: /var/lib/rpm

The dbpath entry is used to define the directory in which the RPM database files are stored. It can be overridden by using the --dbpath option on the RPM command line.

B.4.11 defaultdocdir

Default value: /usr/doc

The defaultdocdir entry is used to define the directory in which RPM will store documentation for all installed packages. It is used only during builds to support the %doc directive.

B.4.12 distribution

Default value: (none)

The `distribution` entry is used to define the distribution for each package. The distribution can also be set by adding the `distribution` tag to a particular spec file. The `distribution` tag in the spec file overrides the `distribution` `rpmrc` file entry.

B.4.13 `excludedocs`

Default value: `0`

The `excludedocs` entry is used to control if documentation should be written to disk when a package is installed. By default, documentation *is* installed; however, this can be overridden by setting the value of `excludedocs` to `1`. Note also that the `--excludedocs` and `--includedocs` options can be added to the RPM command line to override the `excludedocs` entry's behavior. For more information on the `--excludedocs` and `--includedocs` options, please refer to chapter 2 on page 15.

B.4.14 `ftpport`

Default value: (none)

The `ftpport` entry is used to define the port RPM should use when manipulating package files via FTP. See section 2.4.15 on page 33 for more information on how FTP ports are used by RPM.

B.4.15 `ftpproxy`

Default value: (none)

The `ftpproxy` entry is used to define the hostname of the FTP proxy system RPM should use when manipulating package files via FTP. See section 2.4.16 on page 33 for more information on how FTP proxy systems are used by RPM.

B.4.16 `messagelevel`

Default value: 3

The `messagelevel` entry is used to define the desired verbosity level. Levels less than 3 produce greater amounts of output, while levels greater than 3 produce less output.

B.4.17 `netsharedpath`

Default value: (none)

The netsharedpath entry is used to define one or more paths that, on the local system, are shared with other systems. If more than one path is specified, they must be separated with colons.

B.4.18 optflags

Default value: (architecture-specific)

The optflags entry is used to define a standard set of options that can be used during the build process, specifically during compilation.

The format of the optflags entry is slightly different from most other rpmrc file entries. Instead of the usual <name>:<value> format, the optflags entry looks like this:

```
optflags:<architecture> <value>
```

For example, assume the following optflags entries were placed in an rpmrc file:

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

If RPM was running on an Intel 80386-compatible architecture, the optflags value would be set to -O2 -m486 -fno-strength-reduce. If, however, RPM was running on a Sun SPARC-based system, optflags would be set to -O2.

This entry sets the RPM_OPT_FLAGS environment variable, which can be used in the %prep, %build, and %install scripts.

B.4.19 packager

Default value: (none)

The packager entry is used to define the name and contact information for the individual responsible for building the package. The contact information is traditionally defined in the following format:

```
packager:Erik Troan <ewt@redhat.com>
```

B.4.20 pgp_name

Default value: (none)

The pgp_name entry is used to define the name of the PGP public key that will be used to sign each package built. The value is not case sensitive, but the key name entered here must match the actual key name in every other aspect.

For more information on signing packages with PGP, please read chapter 17 on page 241.

B.4.21 pgp_path

Default value: (none)

The pgp_path entry is used to point to a directory containing PGP keyring files. These files will be searched for the public key specified by the pgp_name entry.

For more information on signing packages with PGP, please read chapter 17 on page 241.

B.4.22 require_distribution

Default value: 0

The require_distribution entry is used to direct RPM to require that every package built must contain distribution information. The default value directs RPM to not enforce this requirement. If the entry has a non-zero value, RPM will only build packages that define a distribution.

B.4.23 require_icon

Default value: 0

The require_icon entry is used to direct RPM to require that every package built must contain an icon. The default value directs RPM to not enforce this requirement. If the entry has a non-zero value, RPM will only build packages that contain an icon.

B.4.24 require_vendor

Default value: 0

The require_vendor entry is used to direct RPM to require that every package built must contain vendor information. The default value directs RPM to not enforce this requirement. If the entry has a non-zero value, RPM will only build packages that define a vendor.

B.4.25 rpmdir

Default value: (topdir)/RPMS

The rpmdir entry is used to define the path to the directory in which RPM will write binary package files. Its default value is taken from the value of the topdir entry, with "/RPMS" appended to it. Note that if you redefine `rpmdir`, you'll need to specify a complete path. RPM will automatically add an architecture-specific directory to the end of the path. For example, on an Intel-based system, the actual path would be:

```
/usr/src/redhat/RPMS/i386
```

B.4.26 signature

Default value: (none)

The `signature` entry is used to define the type of signature that is to be added to each package built. At the present time, only signatures from PGP are supported. Therefore, the only acceptable value is "`pgp`".

For more information on signing packages with PGP, please read chapter 17 on page 241.

B.4.27 sourcedir

Default value: `(topdir)/SOURCES`

The `sourcedir` entry is used to define the path to the directory in which RPM will look for sources. Its default value is taken from the value of the `topdir` entry, with "`/SOURCES`" appended to it. Note that if you redefine `sourcedir`, you'll need to specify a complete path.

B.4.28 specdir

Default value: `(topdir)/SPECS`

The `specdir` entry is used to define the path to the directory in which RPM will look for spec files. Its default value is taken from the value of the `topdir` entry, with "`/SPECS`" appended to it. Note that if you redefine `specdir`, you'll need to specify a complete path.

B.4.29 srcrpmdir

Default value: `(topdir)/SRPMS`

The `srcrpmdir` entry is used to define the path to the directory in which RPM will write source package files. Its default value is taken from the value of the `topdir` entry, with "`/SRPMS`" appended to it. Note that if you redefine `srcrpmdir`, you'll need to specify a complete path.

B.4.30 timecheck

Default value: (none)

The timecheck entry is used to define the default number of seconds to apply to the --timecheck option when building packages. For more information on the --timecheck option, please see section 12.1.14 on page 159.

B.4.31 tmppath

Default value: /var/tmp

The tmpdir entry is used to define a path to the directory that RPM will use for temporary work space. This normally consists of temporary scripts that are used during the build process. It should be set to an absolute path (ie, starting with "/").

B.4.32 topdir

Default value: /usr/src/redhat

The topdir entry is used to define the path to the top-level directory in RPM's build directory tree. It should be set to an absolute path (ie, starting with "/"). The following entries base their default values on the value of topdir:

- builddir
- rpmdir
- sourcedir
- specdir
- srcrpmdir

B.4.33 vendor

Default value: (none)

The vendor entry is used to define the name of the organization that is responsible for distributing the packaged software. Normally, this would be the name of a business or other such entity.

Appendix C

Concise RPM Command Reference

C.1 Global Options

The following options can be used in any of RPM's modes:

- --quiet — Print as little output as possible.
- -v — Be a little more verbose.
- -vv — Be incredibly verbose (for debugging).
- --root <dir> — Use <dir> as the top level directory.
- --dbpath <dir> — Use <dir> as the directory for the database.
- --rcfile <file> — Use <file> instead of /etc/rpmrc and $HOME/.rpmrc.

C.2 Informational Options

The following options are used to display information about RPM:
Format: rpm <option>

- --version — Print the version of rpm being used.
- --help — Print a help message.
- --showrc — Show rcfile information.
- --querytags — List the tags that can be used with --queryformat.

C.3 Query Mode

RPM's query mode is used to display information about packages:

Format: rpm --query <options>
or
Format: rpm -q <options>

C.3.1 Package Specification Options To Query Mode

No more than one of the following options may be present in every query command. They are used to select the source of the information to be displayed.

- <packagename> — Query the named package.
- -a — Query all packages.
- -f <file>+ — Query package owning <file>.
- -g <group>+ — Query packages with group <group>.
- -p <packagefile>+ — Query (uninstalled) package <packagefile>.
- --whatprovides <i> — Query packages that provide <i> capability.
- --whatrequires <i> — Query packages that require <i> capability.

C.3.2 Information Selection Options To Query Mode

One or more of the following options may be added to any query command. They are used to select what information RPM will display. If no information selection option is present on the command line, RPM will simply display the applicable package label(s):

- -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).
- --dump — Show all available information for each file (must be used with -l, -c, or -d).
- --provides — List capabilities that the package provides.
- --requires, -R — List capabilities that the package requires.

- `--scripts` — Print the various [un]install, verification scripts.
- `--queryformat <s>` — Use `<s>` as the header format (implies `-i`).
- `--qf <s>` — Shorthand for `--queryformat`.

C.4 Verify Mode

RPM's verification mode is used to ensure that a package is still installed properly:

Format: `rpm --verify <options>`
or
Format: `rpm -V <options>`
or
Format: `rpm -y <options>`

C.4.1 Options To Verify Mode

The following options can be used on any verify command:

- `--nodeps` — Do not verify package dependencies.
- `--nofiles` — Do not verify file attributes.
- `--noscripts` — Do not execute the package's verification script.

C.5 Install Mode

RPM's installation mode is used to install packages:

Format: `rpm --install <packagefile>`
or
Format: `rpm -i <packagefile>`

C.5.1 Options To Install Mode

The following options can be used on any install command:

- `-h`, `--hash` — Print hash marks as package installs (good with `-v`).
- `--prefix <dir>` — Relocate the package to `<dir>`, if relocatable.
- `--excludedocs` — Do not install documentation.

- `--force` — Shorthand for `--replacepkgs` and `--replacefiles`.
- `--ignorearch` — Do not verify package architecture.
- `--ignoreos` — Do not verify package operating system.
- `--includedocs` — Install documentation.
- `--nodeps` — Do not check package dependencies.
- `--noscripts` — Do not execute any installation scripts.
- `--percent` — Print percentages as package installs.
- `--replacefiles` — Install even if the package replaces installed files.
- `--replacepkgs` — Reinstall if the package is already present.
- `--test` — Do not install, but tell if it would work or not.

C.6 Upgrade Mode

RPM's upgrade mode is used to upgrade packages:

Format: `rpm --upgrade <packagefile>`
or
Format: `rpm -U <packagefile>`

C.6.1 Options To Upgrade Mode

The following options can be used on any upgrade command:

- `-h`, `--hash` — Print hash marks as package installs (good with `-v`).
- `--prefix <dir>` — Relocate the package to `<dir>`, if relocatable.
- `--excludedocs` — Do not install documentation.
- `--force` — Shorthand for `--replacepkgs`, `--replacefiles`, and `--oldpackage`.
- `--ignorearch` — Do not verify package architecture.
- `--ignoreos` — Do not verify package operating system.
- `--includedocs` — Install documentation.
- `--nodeps` — Do not verify package dependencies.
- `--noscripts` — Do not execute any installation scripts.

- `--percent` — Print percentages as package installs.
- `--replacefiles` — Install even if the package replaces installed files.
- `--replacepkgs` — Reinstall if the package is already present.
- `--test` — Do not install, but tell if it would work or not.
- `--oldpackage` — Upgrade to an old version of the package (`--force` on upgrades does this automatically).

C.7 Erase Mode

RPM's erase mode is used to erase previously installed packages:

Format: `rpm --erase <package>`
or
Format: `rpm -e <package>`

C.7.1 Options To Erase Mode

The following options can be used on any erase command:

- `--nodeps` — Do not verify package dependencies.
- `--noscripts` — Do not execute any installation scripts.

C.8 Build Mode

RPM's build mode is used to build packages:

Format: `rpm -b<stage> <options> <specfile>`

(Note that `-vv` is the default for all build mode commands.)

C.8.1 Build Mode Stages

One of the following stages must follow the `-b` option:

- `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 — Binary/source package (prep, compile, install, package).

C.8.2 Options To Build Mode

The following options can be used on any build command:

- --short-circuit — Skip straight to specified stage (only for c and i).
- --clean — Remove build tree when done.
- --sign — Generate PGP signature.
- --buildroot <s> — Use <s> as the build root.
- --buildarch <s> — Use <s> as the build architecture.
- --buildos <s> — Use <s> as the build operating system.
- --test — Do not execute any stages.
- --timecheck <s> — Set the time check to <s> seconds (0 disables it).

C.9 Rebuild Mode

RPM's rebuild mode is used to rebuild packages from a source package file. The source archives, patches, and icons that comprise the source package are removed after the binary package is built. Rebuild mode implies --clean.

Format: rpm --rebuild <options> <source-package>

(Note that -vv is the default for all rebuild mode commands.)

C.9.1 Options To Rebuild Mode

Only the global options may be used.

C.10 Recompile Mode

RPM's recompile mode is used to recompile software from a source package file. Unlike --rebuild, no binary package is created.

Format: rpm --recompile <options> <source-package>

(Note that -vv is the default for all recompile mode commands.)

C.10.1 Options To Recompile Mode

Only the global options may be used.

C.11 Resign Mode

RPM's resign mode is used to replace a package's signature with a new one:

Format: `rpm --resign <options> <packagefile>+`

C.11.1 Options To Resign Mode

Only the global options may be used.

C.12 Add Signature Mode

RPM's add signature mode is used to add a signature to a package:

Format: `rpm --addsign <options> <packagefile>+`

C.12.1 Options To Add Signature Mode

Only the global options may be used.

C.13 Check Signature Mode

RPM's check signature mode is used to verify a package's signature:

Format: `rpm --checksig <options> <packagefile>+`
or
Format: `rpm -K <options> <packagefile>+`

C.13.1 Options To Check Signature Mode

The following option can be used on any check signature command:

- `--nopgp` — Skip any PGP signatures (size and MD5 only).

C.14 Initalize Database Mode

RPM's initalize database mode is used to create a new RPM database:

Format: `rpm --initdb <options>`

C.14.1 Options to Initalize database Mode

Only the global options may be used.

C.15 Rebuild Database Mode

RPM's rebuild database mode is used to rebuild an RPM database:

Format: `rpm --rebuilddb <options>`

C.15.1 Options to Rebuild Database Mode

Only the global options may be used.

Appendix D

Available Tags For --queryformat

The following tags were defined at the time this book was written. For the latest list of available queryformat tags, please issue the following command:

```
rpm --querytags
```

Keep in mind that the list of tags produced by the `--querytags` option is the complete list of all tags used by RPM internally; for instance, during package builds. Because of this, some tags do not produce meaningful output when used in a `--queryformat` format string.

D.1 List of --queryformat Tags

For every tag in this section, there can be as many as three different pieces of information:

1. A short description of the tag.
2. Whether the data specified by the tag is an array, and if so, how many members are present in the array.
3. What modifiers can be used with the tag.

The NAME Tag

The `NAME` tag is used to display the name of the package.

Array: No

Used with modifiers: N/A

The VERSION Tag

The VERSION tag is used to display the version of the packaged software.

Array: No

Used with modifiers: N/A

The RELEASE Tag

The RELEASE tag is used to display the release number of the package.

Array: No

Used with modifiers: N/A

The SERIAL Tag

The SERIAL tag is used to display the serial number of the package.

Array: No

Used with modifiers: N/A

The SUMMARY Tag

The SUMMARY tag is used to display a one-line summation of the packaged software.

Array: No

Used with modifiers: N/A

The DESCRIPTION Tag

The DESCRIPTION tag is used to display a detailed summation of the packaged software.

Array: No

Used with modifiers: N/A

The BUILDTIME Tag

The BUILDTIME tag is used to display the time and date the package was created.

Array: No

Used with modifiers: :date

The BUILDHOST Tag

The BUILDHOST tag is used to display the hostname of the system that built the package.

Array: No

Used with modifiers: N/A

The INSTALLTIME Tag

The INSTALLTIME tag is used to display the time and date the package was installed.

Array: No

Used with modifiers: :date

The SIZE Tag

The SIZE tag is used to display the total size, in bytes, of every file installed by this package.

Array: No

Used with modifiers: N/A

The DISTRIBUTION Tag

The DISTRIBUTION tag is used to display the distribution this package is a part of.

Array: No

Used with modifiers: N/A

The VENDOR Tag

The VENDOR tag is used to display the organization responsible for marketing the package.

Array: No

Used with modifiers: N/A

The GIF Tag

The GIF tag is not available for use with --queryformat.

The XPM Tag

The XPM tag is not available for use with --queryformat.

The COPYRIGHT Tag

The COPYRIGHT tag is used to display the copyright terms of the package.

Array: No

Used with modifiers: N/A

The `PACKAGER` Tag

The `PACKAGER` tag is used to display the person or persons responsible for creating the package.

Array: No

Used with modifiers: N/A

The `GROUP` Tag

The `GROUP` tag is used to display the group to which the package belongs.

Array: No

Used with modifiers: N/A

The `CHANGELOG` Tag

The `CHANGELOG` tag is reserved for a future version of RPM.

The `SOURCE` Tag

The `SOURCE` tag is used to display the source archives contained in the source package file.

Array: Yes (Size: One entry per source)

Used with modifiers: N/A

The `PATCH` Tag

The `PATCH` tag is used to display the patch files contained in the source package file.

Array: Yes (Size: One entry per patch)

Used with modifiers: N/A

The `URL` Tag

The `URL` tag is used to display the Uniform Resource Locator that points to additional information on the packaged software.

Array: No

Used with modifiers: N/A

The `OS` Tag

The `OS` tag is used to display the operating system for which the package was built.

Array: No

Used with modifiers: N/A

The ARCH Tag

The ARCH tag is used to display the architecture for which the package was built.
Array: No
Used with modifiers: N/A

The PREIN Tag

The PREIN tag is used to display the package's pre-install script.
Array: No
Used with modifiers: N/A

The POSTIN Tag

The POSTIN tag is used to display the package's post-install script.
Array: No
Used with modifiers: N/A

The PREUN Tag

The PREUN tag is used to display the package's pre-uninstall script.
Array: No
Used with modifiers: N/A

The POSTUN Tag

The POSTUN tag is used to display the package's post-uninstall script.
Array: No
Used with modifiers: N/A

The FILENAMES Tag

The FILENAMES tag is used to display the names of the files that comprise the package.
Array: Yes (Size: One entry per file)
Used with modifiers: N/A

The FILESIZES Tag

The FILESIZES tag is used to display the size, in bytes, of each of the files that comprise the package.

Array: Yes (Size: One entry per file)

Used with modifiers: N/A

The FILESTATES Tag

The FILESTATES tag is used to display the state of each of the files that comprise the package.

Array: Yes (Size: One entry per file)

Used with modifiers: N/A[1]

The FILEMODES Tag

The FILEMODES tag is used to display the permissions of each of the files that comprise the package.

Array: Yes (Size: One entry per file)

Used with modifiers: :perms

The FILEUIDS Tag

The FILEUIDS tag is used to display the user ID, in numeric form, of each of the files that comprise the package.

Array: Yes (Size: One entry per file)

Used with modifiers: N/A

The FILEGIDS Tag

The FILEGIDS tag is used to display the group ID, in numeric form, of each of the files that comprise the package.

Array: Yes (Size: One entry per file)

Used with modifiers: N/A

[1] Since there is no modifier to display the file states in human-readable form, it will be necessary to manually interpret the flag values, based on the RPMFILE_STATE_*xxx* #defines contained in rpmlib.h. This file is part of the rpm-devel package and is also present in the RPM source package.

The `FILERDEVS` Tag

The `FILERDEVS` tag is used to display the major and minor numbers for each of the files that comprise the package. It will only be non-zero for device special files.

Array: Yes (Size: One entry per file)

Used with modifiers: N/A

The `FILEMTIMES` Tag

The `FILEMTIMES` tag is used to display the modification time and date for each of the files that comprise the package.

Array: Yes (Size: One entry per file)

Used with modifiers: `:date`

The `FILEMD5S` Tag

The `FILEMD5S` tag is used to display the MD5 checksum for each of the files that comprise the package.

Array: Yes (Size: One entry per file)

Used with modifiers: N/A

The `FILELINKTOS` Tag

The `FILELINKTOS` tag is used to display the link string for symlinks.

Array: Yes (Size: One entry per file)

Used with modifiers: N/A

The `FILEFLAGS` Tag

The `FILEFLAGS` tag is used to indicate whether the files that comprise the package have been flagged as being documentation or configuration.

Array: Yes (Size: One entry per file)

Used with modifiers: `:fflags`

The `ROOT` Tag

The `ROOT` tag is not available for use with `--queryformat`.

The FILEUSERNAME Tag

The FILEUSERNAME tag is used to display the owner, in alphanumeric form, of each of the files that comprise the package.

Array: No

Used with modifiers: N/A

The FILEGROUPNAME Tag

The FILEGROUPNAME tag is used to display the group, in alphanumeric form, of each of the files that comprise the package.

Array: Yes (Size: One entry per file)

Used with modifiers: N/A

The EXCLUDE Tag

The EXCLUDE tag is deprecated and should no longer be used.

The EXCLUSIVE Tag

The EXCLUSIVE tag is deprecated and should no longer be used.

The ICON Tag

The ICON tag is not available for use with --queryformat.

The SOURCERPM Tag

The SOURCERPM tag is used to display the name of the source package from which this binary package was built.

Array: No

Used with modifiers: N/A

The FILEVERIFYFLAGS Tag

The FILEVERIFYFLAGS tag is used to display the numeric value of the file verification flags for each of the files that comprise the package.

Array: Yes (Size: One entry per file)

Used with modifiers: N/A[2]

[2]Since there is no modifier to display the verification flags in human-readable form, it will be necessary to manually interpret the flag values, based on the RPMVERIFY_*xxx* #defines contained in rpmlib.h. This

The ARCHIVESIZE Tag

The ARCHIVESIZE tag is used to display the size, in bytes, of the archive portion of the original package file.

Array: No

Used with modifiers: N/A

The PROVIDES Tag

The PROVIDES tag is used to display the capabilities the package provides.

Array: Yes (Size: One entry per provide)

Used with modifiers: N/A

The REQUIREFLAGS Tag

The REQUIREFLAGS tag is used to display the requirement flags for each capability the package requires.

Array: Yes (Size: One entry per require)

Used with modifiers: :depflags

The REQUIRENAME Tag

The REQUIRENAME tag is used to display the capabilities the package requires.

Array: Yes (Size: One entry per require)

Used with modifiers: N/A

The REQUIREVERSION Tag

The REQUIREVERSION tag is used to display the version-related aspect of each capbility the package requires.

Array: Yes (Size: One entry per require)

Used with modifiers: N/A

The NOSOURCE Tag

The NOSOURCE tag is used to display the source archives that are not contained in the source package file.

Array: Yes (Size: One entry per nosource)

Used with modifiers: N/A

---

file is part of the rpm-devel package and is also present in the RPM source package.

The NOPATCH Tag

The NOPATCH tag is used to display the patch files that are not contained in the source package file.

Array: Yes (Size: One entry per nopatch)

Used with modifiers: N/A

The CONFLICTFLAGS Tag

The CONFLICTFLAGS tag is used to display the conflict flags for each capability the package conflicts with.

Array: Yes (Size: One entry per conflict)

Used with modifiers: `:depflags`

The CONFLICTNAME Tag

The CONFLICTNAME tag is used to display the capabilities that the package conflicts with.

Array: Yes (Size: One entry per conflict)

Used with modifiers: N/A

The CONFLICTVERSION Tag

The CONFLICTVERSION tag is used to display the version-related aspect of each capability the package conflicts with.

Array: Yes (Size: One entry per conflict)

Used with modifiers: N/A

The DEFAULTPREFIX Tag

The DEFAULTPREFIX tag is used to display the path that will, by default, be used to install a relocatable package.

Array: No

Used with modifiers: N/A

The BUILDROOT Tag

The BUILDROOT tag is not available for use with `--queryformat`.

The INSTALLPREFIX Tag

The INSTALLPREFIX tag is used to display the actual path used when a relocatable package was installed.

Array: No

Used with modifiers: N/A

The EXCLUDEARCH Tag

The EXCLUDEARCH tag is used to display the architectures that should not install this package.

Array: Yes (Size: One entry per excludearch)

Used with modifiers: N/A

The EXCLUDEOS Tag

The EXCLUDEOS tag is used to display the operating systems that should not install this package.

Array: Yes (Size: One entry per excludeos)

Used with modifiers: N/A

The EXCLUSIVEARCH Tag

The EXCLUSIVEARCH tag is used to display the architectures that are the only ones that should install this package.

Array: Yes (Size: One entry per exclusivearch)

Used with modifiers: N/A

The EXCLUSIVEOS Tag

The EXCLUSIVEOS tag is used to display the operating systems that are the only one that should install this package.

Array: Yes (Size: One entry per exclusiveos)

Used with modifiers: N/A

The AUTOREQPROV Tag

The AUTOREQPROV tag is not available for use with `--queryformat`.

The RPMVERSION Tag

The RPMVERSION tag is used to display the version of RPM that was used to build the package.

Array: No

Used with modifiers: N/A

The TRIGGERSCRIPTS Tag

The TRIGGERSCRIPTS tag is reserved for a future version of RPM.

The TRIGGERNAME Tag

The TRIGGERNAME tag is reserved for a future version of RPM.

The TRIGGERVERSION Tag

The TRIGGERVERSION tag is reserved for a future version of RPM.

The TRIGGERFLAGS Tag

The TRIGGERFLAGS tag is reserved for a future version of RPM.

The TRIGGERINDEX Tag

The TRIGGERINDEX tag is reserved for a future version of RPM.

The VERIFYSCRIPT Tag

The VERIFYSCRIPT tag is used to display the script to be used for package verification.

Array: No

Used with modifiers: N/A

Appendix E

Concise Spec File Reference

E.1 Comments

Comments are a way to make RPM ignore a line in the spec file. To create a comment, enter an octothorp (`#`) at the start of the line. Any text following the comment character will be ignored by RPM:

```
# This is the spec file for playmidi 2.3...
```

Comments can be placed in any section of the spec file.

See also: Section 13.1, page 165

E.2 The Preamble

E.2.1 Package Naming Tags

The `name` Tag

The `name` tag is used to define the name of the software being packaged.

```
Name:  cdplayer
```

See also: Section 13.2.1, page 167

The `version` Tag

The `version` tag defines the version of the software being packaged.

```
Version:  1.2
```

See also: Section 13.2.1, page 167

The `release` Tag

The `release` tag can be thought of as the *package's* version.

```
Release: 5
```

See also: Section 13.2.1, page 167

E.2.2 Descriptive Tags

The `%description` Tag

The `%description` tag is used to define an in-depth description of the packaged software. In the descriptive text, a space in the first column indicates that that line of text should be presented to user as-is, with no formatting done by RPM. Blank lines in the descriptive text denote paragraphs.

```
%description
 It slices!
 It dices!
 It's a CD player app that can't be beat.

By using the resonant frequency of the CD itself, it is able to simulate
20X oversampling.  This leads to sound quality that cannot be equaled with
more mundane software...
```

The `%description` tag can be made specific to a particular subpackage by adding the subpackage name, and optionally, the `-n` option:

```
%description bar
```

```
%description -n bar
```

The subpackage name and usage of the `-n` option must match those defined with the `%package` directive.

See also: Section 13.2.2, page 168

The `summary` Tag

The `summary` tag is used to define a one-line description of the packaged software.

```
Summary: A CD player app that rocks!
```

See also: Section 13.2.2, page 169

The `copyright` Tag

The `copyright` tag is used to define the copyright terms applicable to the software being packaged.

```
Copyright: GPL
```

See also: Section 13.2.2, page 169

The `distribution` Tag

The `distribution` tag is used to define a group of packages, of which this package is a part.

```
Distribution: Doors '95
```

See also: Section 13.2.2, page 169

The `icon` Tag

The `icon` tag is used to name a file containing an icon representing the packaged software. The file may be in either GIF or XPM format, although XPM is preferred. In either case, the background of the icon should be transparent.

```
Icon: foo.xpm
```

See also: Section 13.2.2, page 169

The `vendor` Tag

The `vendor` tag is used to define the name of the entity that is responsible for packaging the software.

```
Vendor: White Socks Software, Inc.
```

See also: Section 13.2.2, page 170

The `url` Tag

The `url` tag is used to define a Uniform Resource Locator that can be used to obtain additional information about the packaged software.

```
URL: http://www.gnomovision.com/cdplayer.html
```

See also: Section 13.2.2, page 170

The `group` Tag

The `group` tag is used to group packages together by the types of functionality they provide.

```
Group: Applications/Editors
```

See also: Section 13.2.2, page 170

The `packager` Tag

The `packager` tag is used to hold the name and contact information for the person or persons who built the package.

```
Packager: Fred Foonly <fred@gnomovision.com>
```

See also: Section 13.2.2, page 171

E.2.3 Dependency Tags

The `provides` Tag

The `provides` tag is used to specify a "virtual package" that the packaged software makes available when it is installed.

```
Provides: module-info
```

See also: Section 13.2.3, page 171

The `requires` Tag

The `requires` tag is used to alert RPM to the fact that the package needs to have certain capabilities available in order to operate properly.

```
Requires: playmidi
```

A version may be specified, following the package specification. The following comparison operators may be placed between the package and version:

<, >, =, >=, or <=

```
Requires: playmidi >= 2.3
```

If the `Requires` tag needs to perform a comparison against a serial numbered defined with the `serial` tag, then the proper format would be:

```
Requires: playmidi =S 4
```

See also: Section 13.2.3, page 171

The `serial` Tag

The `serial` tag is used to define a serial number for a package. This is only necessary if RPM is unable to determine the ordering of a package's version numbers.

```
Serial: 4
```

See also: Section 13.2.3, page 172

The `conflicts` Tag

The `conflicts` tag is used to alert RPM to the fact that the package is not compatible with other packages.

```
Conflicts: playmidi
```

A version may be specified, following the package specification. The following comparison operators may be placed between the package and version:

`<`, `>`, `=`, `>=`, or `<=`

```
Conflicts: playmidi >= 2.3
```

If the `conflicts` tag needs to perform a comparison against a serial numbered defined with the `serial` tag, then the proper format would be:

```
Conflicts: playmidi =S 4
```

See also: Section 13.2.3, page 172

The `autoreqprov` Tag

The `autoreqprov` tag is used to control the automatic dependency processing performed when the package is being built. To disable automatic dependency processing, add the following line:

```
AutoReqProv: no
```

(The number `0` may be used instead of `no`) Although RPM defaults to performing automatic dependency processing, the effect of the `autoreqprov` tag can be reversed by changing `no` to `yes`. (The number `1` may be used instead of `yes`)

See also: Section 13.2.3, page 173

E.2.4 Architecture- and Operating System-Specific Tags

The `excludearch` Tag

The `excludearch` tag is used to direct RPM to ensure that the package does *not* attempt to build on the excluded architecture(s).

```
ExcludeArch: sparc alpha
```

See also: Section 13.2.4, page 174

The `exclusivearch` Tag

The `exclusivearch` tag is used to direct RPM to ensure the package is *only* built on the specified architecture(s).

```
ExclusiveArch: sparc alpha
```

See also: Section 13.2.4, page 175

The `excludeos` Tag

The `excludeos` tag is used to direct RPM to ensure that the package does *not* attempt to build on the excluded operating system(s).

```
ExcludeOS: linux irix
```

See also: Section 13.2.4, page 175

The `exclusiveos` Tag

The `exclusiveos` tag is used to denote which operating system(s) should *only* be be permitted to build the package.

```
ExclusiveOS: linux
```

See also: Section 13.2.4, page 175

E.2.5 Directory-related Tags

The `prefix` Tag

The `prefix` tag is used to define part of the path RPM will use when installing the package's files. The prefix can be redefined by the user when the package is installed, thereby changing where the package is installed.

```
Prefix: /opt
```

See also: Section 13.2.5, page 176

The `buildroot` Tag

The `buildroot` tag is used to define an alternate build root, where the software will be installed during the build process.

```
BuildRoot: /tmp/cdplayer
```

See also: Section 13.2.5, page 176

E.2.6 Source and Patch Tags

The `source` Tag

The `source` tag is used to define the filename of the sources to be packaged. When there is more than one `source` tag in a spec file, each one must be numbered so they are unique, starting with the number `0`. When there is only one tag, it does not need to be numbered.

By convention, the source filename is usually preceded by a URL pointing to the location of the original sources, but RPM does not require this.

```
Source0: ftp://ftp.gnomovision.com/pub/cdplayer-1.0.tgz
Source1: foo.tgz
```

See also: Section 13.2.6, page 177

The `nosource` Tag

The `nosource` tag is used to alert RPM to the fact that one or more source files should be excluded from the source package file. The tag is followed by one or more numbers. The numbers correspond to the numbers following the `source` tags that are to be excluded from packaging.

```
NoSource: 0, 3
```

See also: Section 13.2.6, page 177

The `patch` Tag

The `patch` tag is used to define the name of a patch file to be applied to the package's sources. When there is more than one `patch` tag in a spec file, each one must be numbered so they are unique, starting with the number `0`. When there is only one tag, it does not need to be numbered.

```
Patch: cdp-0.33-fsstnd.patch
```

See also: Section 13.2.6, page 178

The `nopatch` Tag

The `nopatch` tag is used to alert RPM to the fact that one or more patch files should be excluded from the source package file. The tag is followed by one or more numbers. The numbers correspond to the numbers following the `patch` tags that are to be excluded from packaging.

```
NoPatch: 2 3
```

See also: Section 13.2.6, page 179

E.3 Scripts

E.3.1 Build-time Scripts

Every build-time script has the following environment variables defined:

- `RPM_SOURCE_DIR`

- RPM_BUILD_DIR
- RPM_DOC_DIR
- RPM_OPT_FLAGS
- RPM_ARCH
- RPM_OS
- RPM_ROOT_DIR
- RPM_BUILD_ROOT
- RPM_PACKAGE_NAME
- RPM_PACKAGE_VERSION
- RPM_PACKAGE_RELEASE

For more information on these environment variables, and build-time scripts in general, please see section 13.3.1 on page 180.

The `%prep` Script

The `%prep` script is the first script RPM executes during a build. As the name implies, it is normally used to prepare the sources for building. The commands in the script can be any valid `sh` commands.

```
%prep
```

See also: Section 13.3.1, page 182

The `%build` Script

The `%build` script is the second script RPM executes during a build. As the name implies, it is normally used to build the software. The commands in the script can be any valid `sh` commands.

```
%build
```

See also: Section 13.3.1, page 182

The `%install` Script

The `%install` script is the third script RPM executes during a build. As the name implies, it is normally used to install the software. The commands in the script can be any valid `sh` commands.

```
%install
```

See also: Section 13.3.1, page 183

The `%clean` Script

The `%clean` script, as the name implies, is used to clean up the software's build directory tree. RPM will normally do this for you, but in certain cases (most notably in those packages that use a build root) you'll need to include a `%clean` script. The commands in the script can be any valid `sh` commands.

`%clean`

See also: Section 13.3.1, page 183

E.3.2 Install-/Erase-time Scripts

These scripts are executed whenever the package is installed or erased. Each script can consist of any valid `sh` commands.

Note: Each of the following scripts can be made specific to a particular subpackage by adding the subpackage name, and optionally, the `-n` option:

`%post bar`

`%preun -n bar`

The subpackage name and usage of the `-n` option must match those defined with the `%package` directive.

Each script has the following environment variable defined:

- `RPM_INSTALL_PREFIX`

For more information on this environment variable please see section 183 on page 183.

The `%pre` Script

The `%pre` script executes just before the package is to be installed.

`%pre`

See also: Section 13.3.2, page 185

The `%post` Script

The `%post` script executes just after the package is to be installed.

`%post`

See also: Section 13.3.2, page 185

The `%preun` Script

The `%preun` script executes just before the package is to be erased.

`%preun`

See also: Section 13.3.2, page 185

The `%postun` Script

The `%postun` script executes just after the package is to be erased.

`%postun`

See also: Section 13.3.2, page 185

E.3.3 Verification Script

The `%verifyscript` Script

The `%verifyscript` script executes whenever the package is verified using RPM's `-V` option. The script can consist of any valid `sh` commands.

See also: Section 13.3.3, page 185

E.4 Macros

E.4.1 The `%setup` Macro

The `%setup` macro is used to unpack the original sources in preparation for the build. It is used in the `%prep` script:

```
%prep
%setup
```

See also: Section 13.4.1, page 186

The `-n` *<name>* Option

The `-n` option is used to set the name of the software's build directory. This is necessary only when the source archive unpacks into a directory named other than `<name>-<version>`.

`%setup -n cd-player`

See also: Section 13.4.1, page 187

The `-c` Option

The `-c` option is used to direct `%setup` to create the top-level build directory before unpacking the sources.

`%setup -c`

See also: Section 13.4.1, page 188

The `-D` Option

The `-D` option is used to direct `%setup` to not delete the build directory prior to unpacking the sources. This option is used when more than one source archive is to be unpacked into the build directory, normally with the `-b` or `-a` options.

```
%setup -D -T -b 3
```

See also: Section 13.4.1, page 189

The `-T` Option

The `-T` option is used to direct `%setup` to not perform the default unpacking of the source archive specified by the first `source` tag. It is used with the `-a` or `-b` options.

```
%setup -D -T -a 1
```

See also: Section 13.4.1, page 189

The `-b <n>` Option

The `-b` option is used to direct `%setup` to unpack the source archive specified on the nth `source` tag line before changing directory into the build directory.

```
%setup -D -T -b 2
```

See also: Section 13.4.1, page 189

The `-a <n>` Option

The `-a` option is used to direct `%setup` to unpack the source archive specified on the nth `source` tag line after changing directory into the build directory.

```
%setup -D -T -a 5
```

See also: Section 13.4.1, page 190

E.4.2 The `%patch` Macro

The `%patch` macro, as its name implies, is used to apply patches to the unpacked sources. With no additional options specified, it will apply the patch file specified by the `patch` (or `patch0`) tag.

```
%patch
```

When there is more than one `patch` tag line in a spec file, they can be specified by appending the number of the `patch` tag to the `%patch` macro name itself.

```
%patch2
```

See also: Section 13.4.2, page 195

The `-P` *<n>* Option

The `-P` option is another method of applying a specific patch. The number from the `patch` tag follows the `-P` option. The following `%patch` macros both apply the patch specified on the `patch2` tag line:

```
%patch -P 2
%patch2
```

See also: Section 13.4.2, page 196

The `-p<#>` Option

The `-p` option is sent directly to the `patch` command. It is followed by a number which specifies the number of leading slashes (and the directories in between) to strip from any filenames present in the patch file.

```
%patch -p2
```

See also: Section 13.4.2, page 196

The `-b` *<name>* Option

When the `patch` command is used to apply a patch, unmodified copies of the files patched are renamed to end with the extension `.orig`. The `-b` option is used to change the extension used by `patch`.

```
%patch -b .fsstnd
```

See also: Section 13.4.2, page 196

The `%patch -E` Option

The `-E` option is sent directly to the `patch` command. It is used to direct `patch` to remove any empty files after the patches have been applied.

See also: Section 13.4.2, page 196

E.5 The `%files` List

The `%files` list indicates which files on the build system are to be packaged. The list consists of one file per line. If a directory is specified, by default all files and subdirectories will be packaged.

```
%files
```

```
/etc/foo.conf
/sbin/foo
/usr/bin/foocmd
```

The `%files` list can be made specific to a particular subpackage by adding the subpackage name, and optionally, the `-n` option:

`%files bar`

`%files -n bar`

The subpackage name and usage of the `-n` option must match those defined with the `%package` directive.

The `%files` list can also use the contents of a file as the list of files to be packaged. This is done by using the `-f` option, which is then followed by a filename:

`%files -f files.list`

See also: Section 13.5, page 198

E.6 Directives For the %files list

E.6.1 File-related Directives

The %doc Directive

The `%doc` directive flags the filename(s) that follow as being documentation.

`%doc README`

See also: Section 13.6.1, page 199

The %config Directive

The `%config` directive is used to flag the specified file as being a configuration file.

`%config /etc/fstab`

See also: Section 13.6.1, page 199

The %attr Directive

The `%attr` directive is used to permit RPM to directly control a file's permissions and ownership. It is normally used when non-root users build packages. The `%attr` directive has the following format:

`%attr(<mode>, <user>, <group>)` *file*

The user and group identifiers must be non-numeric. Attributes that do not need to be set by `%attr` may be replaced with a dash:

`%attr(755, root, -) foo.bar`

See also: Section 13.6.1, page 200

The `%verify` Directive

The `%verify` directive is used to control which of nine different file attributes are to be verified by RPM. The attributes are:

1. `owner` — The file's owner.
2. `group` — The file's group.
3. `mode` — The file's mode.
4. `md5` — The file's MD5 checksum.
5. `size` — The file's size.
6. `maj` — The file's major number.
7. `min` — The file's minor number.
8. `symlink` — The file's symbolic link string.
9. `mtime` — The file's modification time.

If the keyword `not` precedes the list, every attribute *except* those listed will be verified.

```
%verify(mode md5 size maj min symlink mtime) /dev/ttyS0
```

See also: Section 13.6.1, page 200

E.6.2 Directory-related Directives

The `%docdir` Directive

The `%docdir` directive is used to add the specified directory to RPM's internal list of directories containing documentation. When a directory is added to this list, every file packaged in this directory (and any subdirectories) will automatically be marked as documentation.

See also: Section 13.6.2, page 201

The `%dir` Directive

The `%dir` directive is used to direct RPM to package only the directory itself, regardless of what files may reside in the directory at the time the package is created.

```
%dir /usr/blather
```

See also: Section 13.6.2, page 203

E.7 The %package Directive

The `%package` directive is used to control the creation of subpackages. The subpackage name is derived from the first `name` tag in the spec file, followed by the name specified after the `%package` directive. Therefore, if the first `name` tag is:

```
Name:  foo
```

and a subpackage is defined with the following `%package` directive:

```
%package bar
```

the subpackage name will be `foo-bar`.

See also: Section 13.7, page 204

E.7.1 The %package -n Option

The `-n` option is used to change how RPM derives the subpackage name. When the `-n` option is used, the name following the `%package` directive becomes the complete subpackage name. Therefore, if a subpackage is defined with the following `%package` directive:

```
%package -n bar
```

the subpackage name will be `bar`.

See also: Section 13.7.1, page 205

E.8 Conditionals

The `%if`*`xxx`* conditionals are used to begin a section of the spec file that is specific to a particular architecture or operating system. They are followed by one or more architecture or operating system specifiers, each separated by commas or whitespace.

Conditionals may be nested within other conditionals, provided that the inner conditional is completely enclosed by the outer conditional.

The %ifarch Conditional

If the build system's architecture is specified, the part of the spec file following the `%ifarch`, but before a `%else` or `%endif` will be used during the build.

```
%ifarch i386 sparc
```

See also: Section 13.8, page 206

The %ifnarch Conditional

If the build system's architecture is specified, the part of the spec file following the `%ifarch` but before a `%else` or `%endif` will *not* be used during the build.

```
%ifnarch i386 sparc
```

See also: Section 13.8, page 207

The `%ifos` Conditional

If the build system is running one of the specified operating systems, the part of the spec file following the `%ifos` but before a `%else` or `%endif` will be used during the build.

```
%ifos linux
```

See also: Section 13.8, page 207

The `%ifnos` Conditional

If the build system is running one of the specified operating systems, the part of the spec file following the `%ifnos` but before a `%else` or `%endif` will *not* be used during the build.

```
%ifnos linux
```

See also: Section 13.8, page 207

The `%else` Conditional

The `%else` conditional is placed between a `%if` conditional of some persuasion, and an `%endif`. It is used to create two blocks of spec file statements, only one of which will be used in any given case.

```
%ifarch alpha
make RPM_OPT_FLAGS="$RPM_OPT_FLAGS -I ."
%else
make RPM_OPT_FLAGS="$RPM_OPT_FLAGS"
%endif
```

See also: Section 13.8, page 207

The `%endif` Conditional

An `%endif` is used to end a conditional block of spec file statements. The `%endif` is always needed after a conditional, otherwise, the build will fail.

```
%ifarch i386
make INTELFLAG=-DINTEL
%endif
```

See also: Section 13.8, page 208

Appendix F

RPM-related Resources

There are a number of resources available to help you with RPM, over and above the RPM `man` page, and this book. Here are some pointers to them.

F.1 Where to Get RPM

Perhaps before asking, Where can I get RPM? it might be better to see if RPM is already installed on your system. If you have Red Hat Linux on your system, it's there already. But be sure to check on other systems – people are porting RPM to different systems every day, and it just might be there waiting for you.

Here's a quick way to see if RPM is installed on your system:

```
% rpm --version
RPM version 2.3
%
```

If this command doesn't work, it might be that your path doesn't include the directory where RPM resides. Check the usual "binary" directories before declaring RPM a no-show!

F.1.1 FTP Sites

If you can't find RPM on your system, you'll have to grab a copy by FTP. RPM can be found practically anywhere Red Hat Linux is available. While the most obvious site, `ftp.redhat.com`, is certainly an option, it might not be your best choice. For one, it can be *very* busy. For another, unless your link to the Internet is near Red Hat's sprawling development campus in Durham, North Carolina, there's probably an FTP site that is closer to you.

Here is a list of sites that mirror Red Hat Software's main FTP site. Be aware that this list changes frequently, so don't be surprised if a particular site no longer mirrors Red Hat, or has moved it from the paths listed below:[1]

```
FTP Site                Directory
========                =========
sunsite.doc.ic.ac.uk    /packages/linux/redhat

ftp.mpi-sb.mpg.de       /pub/linux/mirror/ftp.redhat.com
ftp.jate.u-szeged.hu    /pub/linux/redhat
ftp.ibp.fr              /pub/linux/distributions/redhat
ftp.gwdg.de             /pub/linux/install/redhat
ftp.msu.ru              /pub/Linux/RedHat
ftp.sgg.ru              /mirror/redhat
sunsite.mff.cuni.cz     /OS/Linux/Distributions/Redhat
ftp.ton.tut.fi          /pub/Linux/RedHat
ftp.funet.fi            /pub/Linux/images/RedHat
sunsite.icm.edu.pl      /pub/Linux/redhat
ftp.arch.pwr.proc.pl    /mirror/linux/redhat
ftp.rhi.hi.is           /pub/linux/RedHat
ftp.nvg.unit.no         /pub/linux/redhat
ftp.pk.edu.pl           /pub/linux/redhat
ftp.nluug.nl            /pub/os/Linux/distr/RedHat
dutepp0.et.tudelft.nl   /pub/Unix/Linux/Distributions/redhat
ftp.iol.ie              /pub/Unix/Linux/distributions/RedHat
sunsite.auc.dk          /pub/os/linux/redhat
ftp.tku.edu.tw          /Unix/Linux/RedHat
ftp.cs.us.es            /pub/Linux/redhat

ftp.is.co.za            /linux/distributions/redhat

ftp.dstc.edu.au         /pub/linux-redhat

ftp.lab.kdd.co.jp       /OS/Linux/packages/redhat
sunsite.ust.hk          /pub/Linux/distributions/redhat

ftp.sunsite.dcc.uchile.cl  /pub/OS/linux/redhat

ftp.interpath.net       /pub/linux/redhat
schlitz.cae.wisc.edu    /pub/Linux/RedHat
ftp.wownet.net          /LINUX/redhat
ftp.engr.uark.edu       /pub/linux/redhat
ftp.infomagic.com       /pub/mirrors/linux/RedHat
ftp.wgs.com             /pub/linux/redhat
ftp.drcdrom.com         /pub/linux-redhat
ftp.hkstar.com          /pub/Linux/redhat
```

[1] An up-to-date copy of the list of mirror sites is always available on `ftp.redhat.com`, in the file `MIRRORS`.

```
ftp.pht.com              /pub/linux/redhat
linux.ucs.indiana.edu    /pub/linux/redhat
ftp.uoknor.edu           /linux/redhat
ftp.cc.gatech.edu        /pub/linux/distributions/redhat
uiarchive.cso.uiuc.edu   /pub/systems/linux/distributions/redhat
ftp.caldera.com          /pub/mirrors/redhat
ftp.cms.uncwil.edu       /linux/redhat
ftp.wilmington.net       /linux/redhat
sunsite.unc.edu          /pub/Linux/distributions/redhat
gatekeeper.dec.com       /pub/linux/redhat
ftp.rge.com              /pub/systems/linux/redhat
linuxwww.db.erau.edu     /pub/linux/distrib/redhat
ftp.eit.com              /pub/mirrors/redhat
ftp.real-time.com        /pub/redhat
```

F.1.2 What Do I Need?

Once you find a nearby site with RPM, and have found the directory where it's kept, you'll notice a variety of files, all starting with "rpm". What are they? Which ones do you need? Here's a representative list, along with the ways in which each file would be used:

```
ftp> ls
200 PORT command successful.
150 Opening ASCII mode data connection for /bin/ls.
total 2689
drwxr-xr-x  6 root  97    2048 Jul 18 10:04 .
drwxr-xr-x  6 root  97    1024 Aug  2 10:09 ..
lrwxrwxrwx  1 root  root    23 Jan 22  1996 RPM-HOWTO.ps ->
                                            ../../docs/RPM-HOWTO.ps
lrwxrwxrwx  1 root  root    24 Jan 22  1996 RPM-HOWTO.txt ->
                                            ../../docs/RPM-HOWTO.txt
-rw-rw-r--  1 root  97   59239 Jan 20  1996 paper.ps.gz
-rw-r--r--  1 root  97  365319 Jul 18 06:05 rpm-2.2.2-1.axp.rpm
-rw-rw-r--  1 root  97  278620 Jul 18 06:05 rpm-2.2.2-1.i386.cpio.gz
-rw-r--r--  1 root  97  282015 Jul 18 06:05 rpm-2.2.2-1.i386.rpm
-rw-r--r--  1 root  97  279855 Jul 18 06:05 rpm-2.2.2-1.sparc.rpm
-rw-r--r--  1 root  97  359354 Jul 18 06:05 rpm-2.2.2-1.src.rpm
-rw-rw-r--  1 root  97  356943 Jul 18 06:05 rpm-2.2.2.tar.gz
-rw-r--r--  1 root  97  122157 Jul 18 06:05 rpm-devel-2.2.2-1.axp.rpm
-rw-r--r--  1 root  97   51132 Jul 18 06:05 rpm-devel-2.2.2-1.i386.rpm
-rw-r--r--  1 root  97   54470 Jul 18 06:05 rpm-devel-2.2.2-1.sparc.rpm
-rw-r--r--  1 root  97   35504 May  1 04:28 rpmbuild.ps.gz
226 Transfer complete.
ftp>
```

Although the version numbers may change, the types of files kept in this directory will not. The files `RPM-HOWTO`, `paper.ps.gz`, and `rpmbuild.ps.gz` contain a variety of information concerning RPM. As such they are valuable sources of supplemental information. The remaining files contain RPM, packaged for various architectures, and in source form. We'll look at them, grouped according to their contents. Here's the first group of files:

```
-rw-r--r--  1 root  97  365319 Jul 18 06:05 rpm-2.2.2-1.axp.rpm
-rw-r--r--  1 root  97  282015 Jul 18 06:05 rpm-2.2.2-1.i386.rpm
-rw-r--r--  1 root  97  279855 Jul 18 06:05 rpm-2.2.2-1.sparc.rpm
```

The files above are the binary package files for RPM version 2.2.2, release 1, on the Digital Alpha, the Intel 386/486/Pentium, and the Sun SPARC. Note that the version number will change in time, but the other parts of the file naming convention won't. As binary package files, they must be installed using RPM. So if you don't have RPM yet, they won't do you much good.[2] Let's look at the next file:

```
-rw-r--r--  1 root  97  359354 Jul 18 06:05 rpm-2.2.2-1.src.rpm
```

This is the source package file for RPM version 2.2.2, release 1. Like the binary packages, the source package requires RPM to install – therefore, it cannot be used to perform an initial install of RPM. Let's see what else there is here:

```
-rw-r--r--  1 root  97  122157 Jul 18 06:05 rpm-devel-2.2.2-1.axp.rpm
-rw-r--r--  1 root  97   51132 Jul 18 06:05 rpm-devel-2.2.2-1.i386.rpm
-rw-r--r--  1 root  97   54470 Jul 18 06:05 rpm-devel-2.2.2-1.sparc.rpm
```

The files above are binary package files that contain the `rpm-devel` subpackage. The `rpm-devel` package contains header files and the RPM library, and is used for developing programs that can perform RPM-related functions. These files cannot be used to get RPM running. That leaves two files left:

```
-rw-rw-r--  1 root  97  278620 Jul 18 06:05 rpm-2.2.2-1.i386.cpio.gz
-rw-rw-r--  1 root  97  356943 Jul 18 06:05 rpm-2.2.2.tar.gz
```

The first file is a gzipped `cpio` archive of the files comprising RPM. After uncompressing the file, `cpio` can be used to extract the files and place them on your system. Note, however, that there is a `cpio` archive for the i386 architecture only. To extract the files, issue the following command:

```
# zcat file.cpio.gz | (cd / ; cpio --extract)
```

[2] If your goal is to install RPM on one of these systems, it might be a good idea to copy the appropriate binary package. That way, once you have RPM running, you can reinstall it with the `--force` option to ensure that RPM is properly installed and configured.

(When actually issuing the command, *file.cpio.gz* should be replaced with the actual name of the `cpio` archive.)

Note that the archive should be extracted using GNU `cpio` version 2.4.1 or greater. It may also be necessary to issue the following command prior to using RPM:

```
# mkdir /var/lib/rpm
```

The last file, `rpm-2.2.2.tar.gz`, contains the sources for RPM. Using it, you can build RPM from scratch. This is the most involved option, but it is the only choice for people interested in porting RPM to a new architecture. See chapter 8 on page 101 for an example of RPM being built from the sources.

F.2 Where to Talk About RPM

As much as we've tried to make this book a comprehensive reference for RPM, there are going to be times when you'll need additional help. The best way to connect with other that use RPM is to try one of the following mailing lists.

F.2.1 The `rpm-list` Mailing List

Red Hat Software, Inc. maintains a mailing list specifically for RPM. In order to subscribe to the list, it's necessary to send a mail message to:

```
rpm-list-request@redhat.com
```

On the message's subject line, place the word `subscribe`. After a short delay, you should receive an automated response with general information about the mailing list.

To send messages to the list, address them to:

```
rpm-list@redhat.com
```

As with other on-line forums, it's advisable to "lurk" for a while before sending anything to the list. That way, you'll be able to see what types of questions are acceptable for the list. Let the list's name be your guide; if the message you want to send doesn't have anything to do with RPM, you shouldn't send it to `rpm-list`!

In general, the flavor of `rpm-list` is a bit biased towards RPM's development, building packages, and issues surrounding the porting of RPM to other systems. If your question is more along the lines of, How do I use RPM to install new software? consider reviewing the first half of this book and lurking on `rpm-list` a while first.

F.2.2 The `redhat-list` Mailing List

The `redhat-list` mailing list is meant to serve as a forum for users of Red Hat Software's Linux operating system. If your questions concerns the use of RPM on Red Hat Linux, then the `redhat-list` is a good place to start. To subscribe, send a message to:

`redhat-list-request@redhat.com`

On the message's subject line, place the word `subscribe`. After a short delay, you should receive an automated response with general information about the mailing list. As with `rpm-list`, it's best to lurk for a while before posting to the list

To send messages to the list, address them to:

`redhat-list@redhat.com`

F.2.3 The `redhat-digest` Mailing List

Some people might find the number of messages on `redhat-list` more than they can handle. However, there is a digest version of the list available. Each digest consists of one or more messages sent to `redhat-list`. The digest is sent out when the collected messages reach a certain size. Therefore, a digest might have one very long message, or twenty smaller ones. In either case, you'll have the collected knowledge of the Red Hat Software development team and their many customers delivered in one message.

To subscribe to `redhat-digest`, send a message to:

`redhat-digest-request@redhat.com`

On the message's subject line, place the word `subscribe`. After a short delay, you should receive an automated response with general information about the mailing list.

To send messages to the list, address them to:

`redhat-list@redhat.com`

As always, observe proper "netiquette" – lurk before you leap!

F.3 RPM On the World Wide Web

Up-to-date information on RPM can always be found at Red Hat Software's web site:

`http://www.redhat.com/`

The site's content changes frequently, so it's impossible to specify an exact URL for RPM information. However, the site is very well run, and always has a comprehensive table of contents as well as a search engine. Either should make finding information on RPM easy.

F.4 RPM's License

RPM is licensed under the GNU General Public License, or as it's more commonly called, the GPL. If you're not familiar with the GPL, it would be worthwhile to spend a few minutes looking it over. The purpose behind the GPL is to ensure that GPL'ed software remains freely available.

"Freely available" doesn't necessarily mean "at no cost," although GPL'ed software is often available by anonymous FTP. The idea behind the GPL is to make it impossible for anyone to take GPL'ed code, and make it proprietary. But enough preliminaries! The best way to understand the GPL is to read it:

GNU GENERAL PUBLIC LICENSE
Version 2, June 1991

F.4.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software – to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps:

1. copyright the software, and
2. offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

F.4.2 GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

   Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

   In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

(c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF

MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

F.4.3 How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program name and a brief idea of what it does.>

               Copyright © 19yy <name of author>

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License as
published by the Free Software Foundation; either version 2 of
the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright © 19yy name of author

Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type
"show w". This is free software, and you are welcome to
redistribute it under certain conditions; type "show c" for
details.
```

The hypothetical commands "show w" and "show c" should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than "show w" and "show c"; they could even be mouse-clicks or menu items – whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.
>
> <signature of Ty Coon>, 1 April 1989
> Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Appendix G

An Introduction to PGP

Assuming you're not the curious type and haven't flipped your way back here, you are probably here looking for some information on the program known as Pretty Good Privacy, or PGP.

G.1 PGP - Privacy for Regular People

PGP, or "Pretty Good Privacy", is a program that is intended to help make electronic mail more secure. It does this by using sophisticated techniques known as *public key encryption*.

If you find yourself wondering what electronic mail and making information unreadable by spies has to do with RPM, you have a good point. However, although PGP's claim to fame is the handling of e-mail in total privacy, it has some other tricks up its sleeve.

G.1.1 Keys your Locksmith Wouldn't Understand

As we mentioned above, PGP uses public key encryption to do some of its magic. You might guess from the name that this type of encryption involves keys of some sort. But, as you might imagine, these are not keys that you can copy down at the local hardware store. They are numbers — really *large* numbers. Here's what a key might look like[1]:


```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: 2.6.2

mQCNAzEpXjUAAAEEAKG4/V9oUSiDc9wIge6Bmg6erDGCLzmFyioAho8kDIJSrcmi
F9qTdPq+fj726pgW1iSb0Y7syZn9Y2lgQm5HkPODfNi8eWyTFSxbr8ygosLRClTP
xqHVhtInGrfZNLoSpv1LdWOme0yOpOQJnghdOMzKXpgf5g84vaUg6PHLopv5AAUR
tCpSZWQgSGF0IFNvZnR3YXJlLCBJbmMuIDxyZWRoYXRAcmVkaGF0LmNvbT6JAFUD
```


[1] When we say that keys are *numbers*, we aren't lying even though the example key doesn't look like a number. It has been processed so that it can be concisely displayed using only printable characters.

```
BRAxc0xcKO2uixUx6ZEBAQ0fAfsGwmueeH3WcjngsAoZyremvyV3Q8C1YmY1EZC9
SWkQxdRKe7n2PY/WiA82Mvc+op1XGTkmqByvxM9Ax/dXh+peiQCVAwUQMXL7xiIS
axFDcvLNAQH5PAP/TdA0yVcuDkXf0PjN/TIjqKRPRt7k6Fm/ameRvzSqB0fMVHEE
5iZKi55Ep1AkBJ3wX257hvduZ/9juKSJjQNuW/FxcHazPU+7yLZmf27xIq7E0ihW
8zz9JNFWSA9+8vlCMBYwdP1a+DzVdwjbJcn0u3/Z/aCY2lYi9U45PzmtU8iJAJUD
BRAxU9GUGX0+IyM0cSUBAbWfA/9+lVfqcpFYkJIV4HuV5niVv7LW4ywxW/SftqCM
lXDXdJdoDbrvLtVYIGWeGwJ6bES6CoQiQjiW7/WaC3BY9ZITQE4hW0PQADz0nZPQ
fdkIIxuIUAUnU/YarasqvxCs5v/TygfWUTPLPSP+MqGqJcDF2UHXCiNAHrItse9M
h7etkYkAdQMFEDEp61/Nq6IpInoskQEB538C+wSIaCNNDOGxlxS5E2tClXRwMYf0
ymuKXs/srvIUjOO7xuIH4K7qcSSdI4eUwuXy6w5tWWR3xZ/XiygcLtKMi2IZIq0j
wmFq7MEk+Xp8MN7Icawkqj1/1p0p4EwKKkIU64kAlQMFEDEp6pZEcVNogr/H7QEB
jp4D/iblfiCzVTA5QhGeW0j1rRxWzohMvnngn29IJgdnN3zuQXB1/lbVV3zYciRH
NyvpynfcTcgORHNpAIxXDaZ7sd48/v7hHLarcR5kxuY0T75X0TG0KT0lFvb4XmcY
HZR2wSWSBteKezB5uK47A6uhwtvPokV0Owk9xPmBV+LPXkW4
=pnqV
-----END PGP PUBLIC KEY BLOCK-----
```


PGP uses two different types of keys: public and private. The public key, as its name suggests, can be shared with anyone. The key shown above is, in fact, a public key. The private key, as *its* name suggests, should be kept a secret. PGP creates keys in pairs — one private and one public. A key pair must remain a pair; if one is lost, the other by itself is useless. Why? Because the two keys have an interesting property that can be exploited in two ways:

- A message encrypted by a given *public* key can only be decrypted with the corresponding *private* key.
- A message encrypted by a given *private* key can only be decrypted with the corresponding *public* key.

In the case of sending messages in total privacy, the key pairs are used in the first manner. It allows two people to exchange private messages without first exchanging any "secret codes". The only requirement is that each know the other's public key.

However, for RPM, the *second* method is the important one. Let's say a company needs to send you a document, and you'd like to make sure it really did come from them. If the company first encrypted the file with their private key and sent it to you, you would have an encrypted file you couldn't read.

Or could you? If you have the company's *public* key, you should be able to decrypt it. In fact, if you can't, you can be sure that the message you received did *not* come from them[2]!

It is this feature that is used by RPM. By using PGP's public key encryption, it is possible to not only prove that a package file came from a certain person or persons, but also that it was not changed somewhere along the line.

[2] Or at least that it didn't make it to you unchanged.

G.1.2 Are RPM Packages Encrypted?

In a word, no. Rather than being encrypted, RPM package files possess a *digital signature*. This is a way of using encryption to attach a signature (again, basically a large number) to some information, such that:

- The signature cannot be separated from the information. Any attempt to verify the signature against any other information will fail.
- The signature can only be produced by one private key.

In the case of RPM, the information being signed is the contents of the .rpm file itself.

A digital signature is just like a regular signature. It doesn't obscure the contents of the document being signed, it just provides a method of determining the authenticity of a document. Here is an example of a digital signature turned into printable text:


```
-----BEGIN PGP SIGNATURE-----
Version: 2.6.3a
Charset: noconv

iQCVAwUBMXVGMFIa2NdXHZJZAQFe4AQAz0FZrHdH8o+zkIvcI/4ABg4gfE7cG0xE
Z2J9GVWD2zi4tG+s1+IWEY6Ae17kx925JKrzF4Ti2upAwTN2Pnb/x0G8WJQVKQzP
mZcD+XNnAaYCqFz8iIuAFVLchYeWj1Pqxxq0weGCtjQIrpzrmGxV7xXzK0jus+6V
rML3TxQSwdA=
=T9Mc
-----END PGP SIGNATURE-----
```


G.1.3 Do All RPM Packages Have Digital Signatures?

Again, no. In a perfect world, every .rpm file would be signed. However, RPM has no formal requirement that this be the case. There is also no requirement that you do anything special with a signed .rpm file. Think of it as an extra feature that you can take advantage of, or not — it's strictly your choice.

G.1.4 So Much to Cover, So Little Time

PGP has a wealth of features, 99% of which we will not cover in this book. For more information on the basics of encryption, *Applied Cryptography*, by Bruce Schneier, contains a wealth of information on the subject. For more details on PGP specifically, O'Reilly's *PGP: Pretty Good Privacy* by Simson Garfinkel is an excellent reference.

If you'd rather surf the 'Net, use your favorite World Wide Web index to hunt for "crypto" or "PGP", and you'll be in business.

G.2 Installing PGP for RPM's Use

To use RPM's PGP-related capabilities, you'll need to have PGP installed on your system. If it's installed already, you should be able to flip to the chapters on verifying package signatures and signing packages and be in business in a matter of minutes. Otherwise, read on for a thumbnail sketch of what's required to install PGP.

G.2.1 Obtaining PGP

The first step in being able to verify .rpm files is to get a copy of PGP. Unfortunately, this is not quite as simple as it might sound. The reason is that PGP is very controversial stuff.

Why the controversy? It centers on PGP's primary mission — to provide a means of communicating with others in complete privacy. As we've discussed, PGP uses encryption to provide this privacy. Good encryption. *Very* good encryption. Encryption so good, it appears some of the world's governments consider PGP a threat to their national security.

Know Your Laws! Various countries have differing stances on the use of "strong encryption" products such as PGP. In some countries, possession of encryption software is strictly forbidden. Other countries attempt to control the flow of encryption technology into (or out of) their country. It is *vital* you know your country's laws, lest you find yourself in prison, or possibly in front of a firing squad!

Patent/Licensing Issues Surrounding PGP Over and above PGP's legal status, there are other aspects to PGP that people living in the U.S. and Canada should keep in mind:

- PGP is free — for non-commercial use only. If you are going to use PGP for business purposes, you should look into getting a commercial copy. PGP is marketed in the United States by:

  Pretty Good Privacy, Inc.
  2121 S. El Camino Real
  Suite 902
  San Mateo, CA 94403
  Main: (415) 572-0430
  Fax: (415) 572-1932
  http://www.pgp.com/

- Part of the software that comprises PGP is protected by several United States patents. Versions of PGP approved for use in the U.S. contain a licensed version of this software, known as RSAREF. RSAREF includes a patent license that allows the use of the software in noncommercial settings only. Commercial use of the technology contained in RSAREF requires a separate license. This is one reason why there are restrictions on the commercial use of PGP in the United States and Canada.

  While people outside the U.S. and Canada can use RSAREF-based PGP, they will probably choose the so-called "international" version. This version replaces RSAREF with software known as MPILIB. MPILIB is, in general, faster than RSAREF, but it cannot legally be used in the United States or Canada.

To summarize, if you are using PGP for commercial purposes in the U.S. or Canada, you'll need to purchase it. Otherwise, people living in the U.S. or Canada should use a version of PGP incorporating RSAREF. People in other countries can use any version of PGP they desire, though they'll probably choose the MPILIB-based "international" version[3].

Getting RSAREF-based PGP The official source for the latest version of PGP based on RSAREF is the Massachusetts Institute of Technology. Due to the restrictions on the export of encryption technology, the process is somewhat convoluted. The easiest way to obtain PGP from the official MIT archive is to use the World Wide Web. Point your web browser at:

`http://web.mit.edu/network/pgp.html`

Simply follow the steps, and you'll have the necessary software on your system in no time.

There is a more cumbersome method that doesn't use the Web. It involves first using anonymous ftp to obtain several files of instructions and license agreements. You will then be directed to use telnet to obtain the name of a temporary ftp directory containing the PGP software. Finally, you can use anonymous ftp to retrieve the software. To start this process, ftp to:

`net-dist.mit.edu`

Then change directory to:

`/pub/PGP`

Obtain a copy of the file `README` and follow the instructions in it *exactly*.

If all this seems like too much trouble, there is another alternative. You can find copies of PGP on just about any BBS, FTP, or Web site advertising freely available software. Be aware, however, that "Floyd's Storm Door and BBS Company" may not be as trustworthy a place as MIT to obtain encryption software. It's really a question of how paranoid you are.

Outside the United States and Canada For people living in other countries, it is much easier to find PGP (depending on the legality of encryption software, of course). Try any of the places you'd normally look for free software. Keep in mind, however, that you shouldn't download PGP from any sites in the U.S. Doing so is considered an "export" of

[3]Note that there are no commercial restrictions regarding PGP in countries other than the U.S. and Canada.

munitions, and can get the people responsible for the site in deep trouble. Wherever you eventually get PGP from, since the patents that complicate matters for the U.S. do not apply abroad, you'll probably end up with the "international" versions of PGP.

G.2.2 Building PGP

Building PGP is mostly a matter of following instructions. However, users of ELF-based Linux distributions (Such as Red Hat Linux) will find that PGP will not build. The problem, according to the PGP FAQ, is that two files do not properly handle the C preprocessor directives that affect support for ELF. The changes are to two files: `80386.S` and `zmatch.S`. Near the beginning of each, you'll find either a `#ifndef` or a `#ifdef` for `SYSV`. If you find:

```
#ifndef SYSV
```

It should be changed to read:

```
#if !defined(SYSV) && !defined(__ELF__)
```

If you find:

```
#ifdef SYSV
```

It should be changed to read:

```
#if defined(SYSV) || defined(__ELF__)
```

After making these changes, PGP should build with no problems.

G.2.3 Ready to Go!

After building and installing PGP, you're ready to start using RPM's package signature capabilities. If your primary interest is in checking the signatures on packages built by someone else, chapter 7 will tell you everything you need to know.

On the other hand, if you are a package builder and would like to start signing packages, chapter 17 will have you signing packages in no time.

Index