RPM HOWTO (RPM at Idle)

Donnie Barnes, djb@redhat.com. Svensk översättning: Linus Åkerlund, uxm165t@tninet.se v2.0, 8 April 1997. Svensk översättning: 8 juni 1998.

Innehåll

1 Inledning

RPM står för **R**ed Hat **P**ackage **M**anager (Red Hat paket-hanterare.övers.anm). Även om namnet innehåller Red Hat, så är RPM helt och hållet avsett att vara ett öppet paket-system, tillgängligt för alla. Det tillåter användare att ta källkoden till ny mjukvara och paketera det till källkods- eller binär-format, så att binär-filerna lätt kan installeras och spåras, och källkod lätt kan kompileras om. Det administrerar även en databas, över alla paket och deras filer, vilken kan användas för att verifiera och "fråga" paket efter information om filer och/eller paket.

Red Hat Software uppmuntrar framställare av andra distributioner att ta sig tid att ta en titt på RPM, och använda det i sina egna distributioner. RPM är ganska flexibelt och lätt-använt, trots att det tillhandahåller en bas för ett väldigt omfattande system. Det är även helt och hållet öppet och tillgängligt, även om de uppskattar bugg-rapporteringar och bugg-fixar. Tillåtelse ges att använda och distribuera RPM, utan avgifter, under GPL.

Mer fullständigt dokumentation om RPM är tillgänglig i Ed Baileys bok *Maximum RPM*. Den är tillgänglig för nedladdning eller beställning från *www.redhat.com* `<http://www.redhat.com>`.

2 Översikt

Låt mig först förklara en del av filosofin bakom RPM. Ett design-mål var att tillåta användningen av "ofördärvad" källkod. Med RPP (vårt tidigare system för paket-administrering, på vilket *inget* i RPM är baserat) var våra källkods-paket "hackade" källkoder, som vi byggde från. Teoretiskt så var det möjligt att installera en källkods-RPM och sedan `make`-a det utan problem. Men källkoden var inte den ursprungliga, och det fanns inga uppgifter om vilka ändringar vi hade gjort, för att få den att kompilera. Man var tvungen att ladda ned den urprungliga källkoden separat. Med RPM får du ofördärvad källkod, tillsammans med en patch, som vi använde för att kompilera från. Vi ser detta som en stor fördel. Varför? Av flera anledningar. En av dem är, att om det kommer en ny version av ett program, så behöver vi inte börja om från början, för att få det att kompilera under RHL. Du kan titta på patchen, för att se vad du *kan* bli tvungen att göra. Alla saker som kompileras in, som standard, är på detta sätt klart synliga.

RPM är också designat att ha kraftfulla "fråge"-möjligheter. Du kan leta igenom hela din paket-databas eller bara vissa filer. Du kan också enkelt ta reda på vilket paket en fil tillhör, och var den kom ifrån. RPM-filerna själva är komprimerade arkiv, men du kan fråga individuella paket enkelt och *snabbt*, på grund av en standardiserad binär rubrik-fil (header file. övers.anm.), vilken innehåller all information du någon gång skulle vilja ha. Denna är inte komprimerad. Detta tillåter *snabb* informations-sökning.

En annan kraftfull funktion är möjligheten att verifiera paket. Om du är orolig att du har raderat en fil som är viktig för något paket, verifiera det bara. Du får ett meddelande om alla ovanligheter. På detta stadium kan du, om det är nödvändigt, ominstallera paketet. Eventuella konfigurerings-filer som du har bevaras också.

Vi vill tacka människorna bakom BOGUS-distributionen för många av deras idéer och koncept, vilka tagits med i RPM. Även om RPM skrivits helt och hållet av Red Hat Software, så är dess funktion baserad på kod som skrivits av BOGUS (PM och PMS).

3 Grundläggande information

3.1 Skaffa RPM

Det bästa sättet att skaffa RPM är att installera Red Hat Linux. Om du inte vill göra det, så kan du fortfarande skaffa och använda RPM. Du kan hämta det från*ftp.redhat.com* `<ftp://ftp.redhat.com/pub/redhat/code/rpm>`.

3.2 RPM-krav

Huvud-kravet för att kunna köra RPM är att du har cpio 2.4.2 eller senare. Även om det här systemet är avsett att användas med Linux, så skulle det mycket väl gå att konvertera det till andra Unix-system. Det har faktiskt kompilerats på SunOS, Solaris, AIX, Irix, AmigaOS och andra operativ-system. Vi måste dock varna dig, för binär-paketen som skapats på en annan Unix-typ kan inte kompileras.

Dessa är de minimala kraven för att installera RPM-paket. För att skapa RPM-paket från källkod behöver du allt som normalt krävs för att kompilera paket, som `gcc`, `make` osv.

4 Använda RPM

I dess enklaste form kan RPM användas för att installera paket:

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

Det näst enklaste kommandot är för att avinstallera ett paket:

```
rpm -e foobar
```

Ett av de mest komplicerade, men *mycket* användbara kommandona, låter dig installera paket via FTP. Om du är uppkopplat på nätet, och vill installera ett nytt paket, så är allt du behöver göra att specificera en korrekt URL, t.ex.:

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

Observera att RPM nu kommer att fråga och/eller installera via FTP.

Även om dessa kommandon är enkla, så kan rpm användas på många olika sätt, vilket vi ser i `Usage`-meddelandet:

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

usage: rpm {--help}
       rpm {--version}
       rpm {--initdb}   [--dbpath <dir>]
       rpm {--install -i} [-v] [--hash -h] [--percent] [--force] [--test]
```

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

Du kan hitta mer detaljer om vad dessa alternativ gör i man-sidan för RPM.

5 Okej, vad kan jag *faktiskt* göra med RPM?

RPM är ett mycket användbart verktyg och, som du ser, så har det många möjligheter. Det bästa sättet att förstå dem är att ta en titt på några exempel. Jag tog upp installering och avinstallering ovan, så här kommer några andra exempel:

- Låt oss säga att du, av misstag, raderat några filer, men inte är säker på vad du har raderat. Om du ville verifiera hela ditt system, och se efter vad som fattas, så skulle du skriva:

  ```
  rpm -Va
  ```

- Låt oss säga att du hittar en fil som du inte känner igen. För att ta reda på vilket paket den tillhör, så skulle du skriva:

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

  Utdatan skulle bli:

```
xjewel-1.6-1
```

- Du hittar en ny koules-RPM, men du vet inte vad det är. För att få en del information skriver du:

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

Utdatan skulle bli:

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

- Vidare vill du se vilka filer som koules-RPM installerar. Du skulle skriva:

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

Utdatan blir:

```
/usr/doc/koules
/usr/doc/koules/ANNOUNCE
/usr/doc/koules/BUGS
/usr/doc/koules/COMPILE.OS2
/usr/doc/koules/COPYING
/usr/doc/koules/Card
/usr/doc/koules/ChangeLog
/usr/doc/koules/INSTALLATION
/usr/doc/koules/Icon.xpm
/usr/doc/koules/Icon2.xpm
/usr/doc/koules/Koules.FAQ
/usr/doc/koules/Koules.xpm
/usr/doc/koules/README
/usr/doc/koules/TODO
/usr/games/koules
/usr/games/koules.svga
/usr/games/koules.tcl
/usr/man/man6/koules.svga.6
```

Dessa är bara några exempel. Mer kreativa exempel kan du komma på när du har lärt känna RPM bättre.

6 Skapa RPM-paket

Att skapa RPM-paket är också ganska lätt, speciellt om du kan få mjukvaran som du försöker göra ett paket av att kompilera.

Den grundläggande proceduren för att skapa ett RPM-paket är som följer:

- Se till att `/etc/rpmrc` är inställd för ditt system.

- Få källkoden som du ska bygga RPM-paketet för att kompilera på ditt system.
- Gör en patch med de ändringar du behövde göra, för att få källkoden att kompilera ordentligt.
- Gör en specifikations-fil för paketet.
- Se till så att allt är på sin plats.
- Skapa paketet med hjälp av RPM

Under normal användning skapar RPM både binär- och källkods-paket.

6.1 rpmrc-filen

Som det är nu, så är det enda sättet att konfigurera RPM via `/etc/rpmrc`-filen. Ett exempel ser ut så här:

```
require_vendor: 1
distribution: I roll my own!
require_distribution: 1
topdir: /usr/src/me
vendor: Mickiesoft
packager:  Mickeysoft Packaging Account <packages@mickiesoft.com>

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

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

tmppath: /usr/tmp
```

`require_vendor`-raden gör att RPM kräver (require) att finna en vendor-rad. Denna kan komma från `/etc/rpmrc` eller från rubrik-delen (header) eller spec-filen själv. För att slå av detta, så kan du ändra värdet till `0`. Detsamma gäller för `require_distribution`- och `require_group`-raderna.

Nästa rad är `distribution`-raden. Du kan ange denna här, eller senare i rubrik-delen av spec-filen. När du skapar ett RPM-paket för en speciell distribution, så är det en god idé att se till att den här raden är korrekt, även om det inte krävs. `vendor`-raden fungerar i stort sett på samma sätt, men kan vara vad som helst (alltså Joe's Software och Rock Music Emporium).

RPM har nu också stöd för att skapa paket på flera olika arkitekturer. `rpmrc`-filen kan innehålla flera ”optflag”-variabler (optimerings-flaggor), för att kompilera saker, som kräver en arkitektur-specifik flagga under kompileringen. Se senare avsnitt för om hur du kan använda denna varibel.

Utöver de ovan nämnda makrona, så finns det flera till. Du kan använda:

```
rpm --showrc
```

för att ta reda på hur dina taggar är satta, och vilka alla tillgängliga flaggor är.

6.2 Spec-filen

Vi ska börja med att diskutera spec-filen. Spec-filer krävs, för att du ska kunna skapa ett paket. Spec-filen är en beskrivning av mjukvaran, tillsammans med instruktioner för hur den ska kompileras, och en fil-lista över alla de binär-filer som installeras.

Du bör ge din spec-fil ett namn, i linje med konventionen. Det ska vara paket-namn-bindestreck-versions-nummer-bindestreck-utgåvonummer-punkt-spec.

Här är en liten spec-fil (vim-3.0-1.spec): (Det står så, men det verkar uppenbart att det rör sig om eject-1.4-3.spec. övers.anm.)

```
Summary: ejects ejectable media and controls auto ejection
Name: eject
Version: 1.4
Release: 3
Copyright: GPL
Group: Utilities/System
Source: sunsite.unc.edu:/pub/Linux/utils/disk-management/eject-1.4.tar.gz
Patch: eject-1.4-make.patch
Patch1: eject-1.4-jaz.patch
%description
This program allows the user to eject media that is autoejecting like
CD-ROMs, Jaz and Zip drives, and floppy drives on SPARC machines.

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

%build
make RPM_OPT_FLAGS="$RPM_OPT_FLAGS"

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

%files
%doc README COPYING ChangeLog

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

6.3 Rubrik-delen

Rubrik-delen innehåller några standard-fält, som du måste fylla i. Det finns några konventioner du måste följa också. Fälten som måste fyllas i är de följande:

- `Summary:` Detta är en en-rads beskrivning av paketet.
- `Name:` Detta måste vara namn-strängen från rpm-filnamnet du har tänkt använda
- `Version:` Detta måste vara versions-strängen från rpm-filnamnet du har tänkt använda.

- `Release:` Detta är utgåvo-numret för ett paket av samma version (alltså, om vi skapar ett paket, och upptäcker något mindre fel i det, så att vi måste skapa om det, så ska nästa paket ha utgåvo-nummer 2).

- `Icon:` Detta är namnet på ikon-filen, för användning med andra hög-nivå installerings-verktyg (som Red Hats "glint"). Den måste vara en gif, och finnas i SOURCES-katalogen.

- `Source:` Denna rad pekar ut den ofördärvade källkods-filens HEM-katalog. Den används om du någonsin skulle vilja ha källkoden igen, eller leta efter nya versioner. Konvention: filnamnet i den här raden MÅSTE matcha filnamnet du har på ditt eget system (ladda alltså inte ned källkoden och ändra dess namn). Du kan också ange mer än en källkodsfil, genom att använda rader som: using lines like:

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

  Dessa filer ska vara i `SOURCES`-katalogen. (Katalog-strukturen diskuteras i ett senare avsnitt, "Källkodens katalog-träd".)

- `Patch:` Detta är stället där du kan hitta patchen, om du skulle behöva ladda ned den igen. Konvention: filnamnet här måste matcha det du använder när du skapar DIN patch. Observera också att du kan ha flera patch-filer, precis som kan ha flera källkods-filer. Du kan ha något i stil med:

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

  Dessa filer ska finnas i `SOURCES`-katalogen.

- `Copyright:` Denna rad anger hur upphovsrätten för paketet ser ut. Du bör använda någonting i stil med GPL, BSD, MIT, public domain, distribuerbart eller kommersiellt.

- `BuildRoot:` Denna rad låter dig specificera en katalog som "rot" för skapandet och installerandet av nya paket. Du kan använda detta för att hjälpa dig att testa ditt paket, innan du installerar det på din maskin.

- `Group:` Denna rad används för att tala om för hög-nivå installerings-program (som Red Hats "glint"), var de ska placera detta speciellt program i den hierarkiska strukturen. Grupp-trädet ser för tillfället ut ungefär så här:

  ```
  Applications
      Communications
      Editors
          Emacs
      Engineering
      Spreadsheets
      Databases
      Graphics
      Networking
      Mail
      Math
      News
      Publishing
          TeX
  Base
      Kernel
  ```

```
Utilities
    Archiving
    Console
    File
    System
    Terminal
    Text
Daemons
Documentation
X11
    XFree86
        Servers
    Applications
        Graphics
        Networking
    Games
        Strategy
        Video
    Amusements
    Utilities
    Libraries
    Window Managers
Libraries
Networking
    Admin
    Daemons
    News
    Utilities
Development
    Debuggers
    Libraries
        Libc
    Languages
        Fortran
        Tcl
    Building
    Version Control
    Tools
Shells
Games
```

- `%description` Detta är inte riktigt en rubrik-post, men bör tas upp i anslutning till resten av rubrikdelen. Du behöver en ”description” per paket och/eller under-paket. Detta är ett fält på flera rader, som du ska använda för att ge en omfattande beskrivning av paketet.

6.4 Prep

Detta är den andra avdelningen i spec-filen. Den används för att göra källkoden redo för kompilering. Här ska du göra allt som är nödvändigt för att få källkoden patchad och inställd, som den behöver vara konfigurerad för att köra `make`.

En sak att observera: Vartenda av dessa avsnitt är egentligen bara ett ställe att köra skal-program. Du skulle helt enkelt kunna göra ett sh-skal-program och stoppa in det efter `%prep`-taggen, för att packa upp och patcha källkoden. Vi har dock skapat makron för att hjälpa till med detta.

Den första av dessa makron är `%setup`-makron. I dess enklaste form (inga kommando-rads-parametrar), så packar den helt enkelt upp källkoden och `cd`-ar till källkods-katalogen. Den känner också till följande parametrar:

- `-n namn` sätter namnet på installerings-katalogen till det angivna `namnet`. Standarden är `$NAMN-$VERSION`. Andra möjligheter inbegriper `$NAMN`, `${NAMN}${VERSION}`, eller vad den huvudsakliga tar-filen nu använder. (Observera att dessa "$"-variabler *inte* är riktiga varibler, som är tillgängliga inom spec-filen. De används här bara i stället för exempel på namn. Du måste använda det riktiga namnet och den riktiga versionen i ditt paket, inte en variabel.)
- `-c` skapar och cd-ar till den angivna katalogen *innan* den kör untar.
- `-b #` packar upp Source# *innan* den cd-ar till katalogen (och detta går inte ihop med `-c`, så använd det inte). Detta är bara användbart i paket med flera källkods-filer.
- `-a #` packar upp Source# *efter* att det cd-ar till katalogen.
- `-T` Denna parameter kör över standard-handlingen, att packa upp källkoden, och kräver en `-b 0` eller `-a 0` för att få den huvudsakliga källkods-filen uppackad. Du behöver detta när det finns sekundära källkodsfiler.
- `-D` Radera *inte* katalogen innan uppackningen. Det är endast användbart där du har mer än en konfigurerings-makro. Den ska *endast* användas i konfigurerings-makron *efter* den första (men aldrig i den första).

Nästa av de tillgängliga makrona är `%patch`-makron. Denna makro hjälper dig att automatisera processen att lägga till patchar till källkoden. Den känner till flera parametrar, vilka räknas upp nedan:

- `#` lägger till Patch# som patch-fil.
- `-p #` anger antalet kataloger att strippa, för patch(1)-kommandot.
- `-P` Standard-handlingen är att lägga till `Patch` (eller `Patch0`). Denna parameter förhindrar detta och kräver en `0` för att få den huvudsakliga källkods-filen uppackad. Denna parameter är användbar i en andra (eller senare) `%patch`-makro, vilken kräver ett annat nummer än den första makron.
- Du kan också köra `%patch#`, istället för att köra det riktiga kommandot: `%patch # -P` .

Fler makron än så bör du inte behöva. Efter att du har gjort allt detta, så kan du göra vilka ytterligare inställningar du behöver, genom skal-program av `sh`-typen. Allt du tar med, fram till `%build`-makron (vilken tas upp i nästa avsnitt) körs via `sh`. Se exemplen ovan för information om vilka sorters saker du kan tänkas vilja göra här.

6.5 Build

Det finns egentligen inga makron för detta avsnitt. Här ska du bara lägga in de kommandon, som du behöver för att kompilera mjukvaran, efter att du har packat upp källkoden, patchat den och cd-at till katalogen. Det här är bara en uppsättning kommandon som skickas till `sh`, så alla giltiga `sh`-kommandon fungerar här (inklusive kommentarer). **Din aktuella arbets-katalog ställs i varje avsnitt om till källkodens topp-katalog**, glöm inte det. Du kan `cd`-a till underkataloger, om det är nödvändigt.

6.6 Install

Det finns inga makron här, heller. Här ska du bara stoppa in de kommandon som är nödvändiga för att installera. Om du har `make install` tillgängligt, i paketet du skapar, lägg in det här. Om inte, så kan du antingen patch makefilen, så att den har `make install` och sedan göra en `make install` här, eller så kan du installera filerna manuellt med `sh`-kommandon. Du kan se din aktuella katalog som källkodens topp-katalog.

6.7 Valfria skal-program som körs före eller efter installering/avinstallering

Du kan lägga in skal-program före och efter installering och avinstalleringen, i binär-paket. En bra anledning att göra detta är för att göra saker som att köra `ldconfig` efter installeringen, eller radera paket som innehåller delade bibliotek (shared libraries). Makrona för alla skal-program är som följer:

- `%pre` är makrot för skal-program som körs innan installeringen.
- `%post` är makrot för skal-program som körs efter installeringen.
- `%preun` är makrot för skal-program som körs före avinstalleringen.
- `%postun` är makrot för skal-program som körs efter avinstalleringen.

Innehållet i dessa avsnitt kan vara vilka `sh`-program som helst, men du behöver *inte* `#!/bin/sh`.

6.8 Files

Det här är avsnittet där du *måste* räkna upp filerna till binär-paketet. RPM kan inte på något sätt veta vilka binär-filer som installeras som ett resultat av `make install`. Det finns *INGET* sätt att göra detta. Vissa har föreslagit att det skulle köra `find` innan och efter att paketet installerats. I ett system med flera användare är detta dock oacceptabelt, eftersom andra filer kan skapas, under det att ett paket skapas, som inte har någonting att göra med själva paketet.

Det finns några makron tillgängliga för att göra några speciella grejer här också. De räknas upp och beskrivs här:

- `%doc` används för att markera dokumentation i källkods-paketet, som du vill installera i en binär-installering. Dokumenten kommer installeras i `/usr/doc/$NAMN-$VERSION-$UTGÅVA`. Du kan räkna upp flera dokument på samma rad med denna makro, eller så kan du räkna upp dem separat, med en makro för varje dokument.
- `%config` används för att markera konfigurerings-filer i ett paket. Detta inkluderar filer som sendmail.cf, passwd osv. Om du senare avinstallerar ett paket som innehåller konfigurerings-filer, så kommer alla oförändrade filer raderas, och de som har ändrats kommer flyttas till sina gamla namn, med `.rpmsave` tillagt till namnet. Du kan räkna upp flera filer med denna makro också.
- `%dir` markerar en enda katalog, i en fillista, så att den blir inkluderad som ”ägd” av paketet. Som standard är det så, att om du listar ett katalog-namn *UTAN* en `%dir`-makro, så kommer *ALLT* i den katalogen att inkluderas i fil-listan, och senare installeras, som en del av det paketet.
- `%files -f <filnamn>` låter dig lista dina filer i en godtyckligt vald fil, i källkodens build-katalog. Det här är trevligt i de fall där du har ett paket som kan skapa sin egen fillista. Då inkluderar du bara den fillistan här, och så behöver du inte själv räkna upp filerna.

Det viktigaste att tänka på i fillistan är hur du räknar upp kataloger. Om du tar med `/urs/bin` av misstag, så kommer ditt binär-paket att innehålla *varje* fil i `/usr/bin` på ditt system.

6.9 Bygga paketet

6.9.1 Källkodens katalog-träd

Det första du behöver är ett korrekt konfigurerar build-katalog-träd. Detta kan du ställa in med `/etc/rpmrc`-filen. De flesta använder helt enkelt `/usr/src`.

Du kan bli tvungen att skapa följande kataloger i ditt build-träd:

- `BUILD` är katalogen där allt skapande (building) utförs av RPM. Du behöver inte utföra dina test-byggen på något speciellt ställe, men det är här RPM kommer utföra sitt byggnads-arbete.
- `SOURCES` är katalogen där du bör stoppa in dina ursprungliga källkods-filer (packade med tar) och dina patchar. Här kommer RPM leta, som standard.
- `SPECS` är katalogen där alla spec-filer ska finnas.
- `RPMS` är katalogen där RPM kommer stoppa de binära RPM-paket som det byggt.
- `SRPMS` är katalogen där alla källkods-RPM-paket kommer hamna.

6.9.2 Test-bygga

Det första du antagligen kommer vilja göra är att få källkoden att kompilera, utan att använda RPM. För att göra detta, packa upp källkoden, och byt ut katalog-namnet till $NAMN.orig. Packa sedan upp källkoden igen. Använd källkoden att kompilera från. Gå in i källkods-katalogen och följ instruktionerna för att kompilera den. Om du måste modifiera saker och ting, så behöver du en patch. Så fort du har kompilerat det, städa upp i källkods-katalogen. Se till så att du har tagit bort alla filer som skapas av `configure`-programmet. `cd`-a ut ur källkods-katalogen till dess föräldra-katalog. Sen kan du göra något i stil med:

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

Detta skapar en patch åt dig, som du kan använda i din spec-fil. Observera att ”linux” i patch-namnet bara är ett sätt att identifiera den. Det är bäst om du använder något mer beskrivande namn, såsom ”config” eller ”bugs” för att beskriva *varför* du var tvungen att skapa en patch. Det är också en god idé att titta på patch-filerna du skapar, innan du använder dem, för att se till så att inga binär-filer kom med av misstag.

6.9.3 Skapa fil-listan

Nu när du har källkoden som ska kompileras, och vet hur du ska göra det, kompilera och installera den. Titta på utdatan från installerings-sekvensen och skapa din fil-lista, som du ska använda i spec-filen, från den. Vi skapar oftast spec-filen parallellt med dessa andra steg. Du kan skapa en första fil-lista och fylla i de enkla delarna, och sedan fylla i de andra stegen, medan du utför dem.

6.9.4 Skapa paketet med RPM

Så fort du har en spec-fil, så är du klar att försöka skapa ditt paket. Det bästa sättet att göra detta, är med en kommando-rad i stil med:

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

Det finns också andra alternativ, som är användbara med `-b`-parametern:

- `p` betyder att endast `prep`-avsnittet i spec-filen ska köras.
- `l` är ett list-kollning, som gör några kontroller på `%files`.
- `c` gör en prep och kompilerar. Detta är användbart när du är osäker på om källkoden alls kommer att kompilera. Det kan verka onödigt, eftersom du helst vill fippla med källkoden själv, tills du får den att kompilera, och sedan använda RPM, men när du vant dig vid att använda RPM, så kommer du hitta tillfällen då du får användning för detta.
- `i` gör en prep, kompilera och installera.
- `b` prep, kompilera, installera och bygg endast ett binär-paket.
- `a` skapa allt (båda källkods- och binär-paket).

Det finns flera alternativ till `-b`-parametern. Dessa är:

- `--short-circuit` hoppar direkt till ett specifikt steg (kan –> –endast användas med c och i).
- `--clean` raderar byggnades-trädet när det allt är klart.
- `--keep-temps` spara alla temporära filer och skal-program, –> –som skapades i /tmp. Du kan faktiskt se vilka filer som skapades i –> –/tmp med `-v`-parametern.
- `--test` utför inga steg på riktigt, men använder keep-temp.

6.10 Testa det

När du har en källkods- och en binär-rpm för ditt paket, så måste du testa det. Det enklaste och bästa sättet är att använda en helt annan maskin än den du skapade paketet på. När allt kommer omkring så har du bara gjort en massa `make install` på din egen maskin, så det borde vara ganska väl installerat.

Du kan köra `rpm -u paketnamn` på paketet du vill testa, men det kan vara förrädiskt, eftersom du, då du skapade paketet, gjorde en `make install`. Om du glömde att ta med något i fil-listan, så kommer det inte bli avinstallerat. Du kommer då om-installera binär-paketet och ditt system kommer vara komplett igen, men din rpm är det fortfarande inte. Kom ihåg, att bara för att du kör `rpm -ba paket`, så kommer de flesta som installerar paketet köra `rpm -i paket`. Se till så att du inte gör något i `build`- och `install`-avsnitten, som behöver göras när binär-filerna redan är installerade.

6.11 Vad du ska göra med dina nya RPM-paket

Så fort du skapat dina egna RPM-paket (om vi förutsätter att det är något som inte redan gjorts till RPM-paket), så kan du dela med dig av ditt arbete till andra (vilket också förutsätter att det du gjorde ett RPM-paket av är fritt distribuerbart). För att göra detta, ladda upp det till *ftp.redhat.com* `<ftp://ftp.redhat.com>`.

6.12 Och nu då?

Se avsnittet ovan, om att Testa det och Vad du ska göra med dina nya RPM-paket. Vi vill ha alla tillgängliga RPM-paket vi kan få, och vi vill att de ska vara bra RPM-paket. Ta dig tid att testa dem ordentligt, och ta dig sedan tid att ladda upp dem, så att alla kan få tillgång till dem. *Var vänlig* se till att du laddar upp *fritt distribuerbar mjukvara*. Kommersiell mjukvara och shareware ska *inte* laddas upp, om de inte har en copyright som explicit säger att det är tillåtet. Detta inkluderar Netscapes mjukvara, ssh, pgp osv.

7 RPM-skapande för flera arkitekturer

RPM kan nu användas för att bygga paket för Intel i386, Digital Alpha som kör Linux och Sparc. Det har även rapporterats att det fungerar på SGIs och HPs arbetsstationer. Det finns flera funktioner som gör det enkelt att skapa paket på alla plattformar. Den första av dessa är ”optflags”-direktivet i `/etc/rpmrc`. Det kan användas för att ange flaggor, vilka används när mjukvara byggs med arkitektur-specifika värden. En annan funktion är ”arch”-makrona i spec-filen. De kan användas för att utföra olika saker, beroende på vilken arkitektur du skapar paketen på. En annan funktion är ”Exclude”-direktivet i rubrik-delen.

7.1 Exempel på spec-filer

Det följande är delar av spec-filen till ”fileutils”-paketet. Det är konfigurerat för att kunna byggas på både Alpha och Intel.

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

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

The ls program in this package now incorporates color ls!

%prep
%setup

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

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

.
.
.
```

7.2 Opt-flaggor

I det här exemplet kan du se hur ”optflags”-direktivet används från `/etc/rpmrc`. Beroende på vilken arkitektur det byggas på, så ges det korrekta värdet till `RPM_OPT_FLAGS`. Du måste patcha Makefilen för ditt paket, för att använda denna variabel, istället för de vanliga direktiven som kanske används (som `-m486` och `-O2`). Du kan få en bättre känsla för vad som behöver göras, genom att installera detta källkods-paket och sedan packa upp källkoden och undersöka Makefilen. Ta sedan en titt på patchen för Makefilen och se vilka ändringar som måste göras.

7.3 Makron

`%ifarch`-makron är väldigt viktig i allt detta. I de flesta fall behöver du skapa en patch eller två, som är specifika för endast en arkitektur. I det här fallet kommer RPM tillåta dig att lägga till den patchen till endast en arkitektur.

I exemplet ovan har fileutils en patch för 64-bitars maskinter. Denna ska uppenbarligen endast läggas till på Alpha. Så vi la till en `%ifarch`-makro kring 64-bitars-patchen, på följande sätt:

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

Detta ser till så att patchen inte läggs till på någon annan arkitektur än alpha.

7.4 Utesluta arkitekturer från paket

För att du ska kunna underhålla källkods-paket i en katalog, för alla plattformar, så har vi implementerat möjligheten att ”utesluta” paket från att byggas på vissa arkitekturer. Detta är för att du fortfarande ska kunna göra saker som

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

och få rätt paket att byggas. Om du fortfarande inte har portat en applikation till en viss plattform, så är allt du behöver göra att lägga till en rad som:

```
ExcludeArch: axp
```

till rubrik-delen av spec-filen i källkods-paketet. Bygg sedan om paketet på plattformen som det ska kunna byggas på. Du har sedan ett källkods-paket som går att bygga på en Intel, och kan som enkelt kan hoppas över på en Alpha.

7.5 Avslutning

Att använda RPM för att bygga paket för flera arkitekturer är oftast enklare att göra än att skaffa paketet självt och bygga det på båda ställena. Detta blir dock mycket enklare, i det att fler av de ”hårda” paketen byggs. Som alltid så är den bästa hjälpen, om du fastnar, då du bygger RPM-paket, att titta på andra, liknande paket.

8 Upphovsrätt