레드햇 패키징 하우투 RPM HOWTO (RPM at Idle)

Donnie Barnes, djb@redhat.com

V2.0, April 8, 1997

번역: 이기동 kidong@shinbiro.com

목 차

제 1 절 소개

RPM은 레드햇 패키지 관리자이다. 여기에는 레드햇이라는 이름이 들어 있지만, 누구나 사용할 수 있는 개방된 패키징 시스템으로 만들어졌다. RPM은 사용자가 새로운 프로그램의 소스 코드를 소스와 바이너리로 패키징이 가능하도록 한다. 이렇게 되면 바이너리를 쉽게 설치하고 찾아내고 소스를 쉽게 다시 빌드할 수 있게 된다. 이것은 모든 패키지와 파일의 데이터베이스를 관리하는데, 이는 패키지를 검증하고 파일과 패키지의 질의에 쓰인다.

레드햇 소프트웨어는 다른 배포본 제작자들이 RPM을 살펴보고 사용하는 배포본에 채용할 것을 권장한다. 이렇게 많은 부문에서 확장 가능한 기반을 제공함에도 불구하고도, RPM은 매우 유연하고 사용하기 쉽다. RPM은 전부 개방되었고 사용가능해서 우리는 버그리포트와 수정을 보내 주는 것에 감사하는 바이다. RPM은 GPL을 따라 로얄티 없이 배포된다.

RPM에 대해 더 완벽한 문서는 Ed Bailey씨가 쓴 *Maximum RPM*이란 책이 있다. 이 책은 다운로드 받을 수 있고 *www.redhat.com* `<http://www.redhat.com>` 에서 구입할 수도 있다.

제 2 절 개요

첫째로, 필자가 RPM을 바탕으로 하는 철학을 설명하고자 한다. 설계한 목적중 하나는 사용자들이 소스를 "그대로" 사용할 수 있도록 하는 것이다. RPP(RPM 이전의 패키징 시스템)로 만든 소스 패키지들은 우리가 빌드한 소스에서 "해킹" 한 것이었다. 이론적으로, 한 사람이 RPP로된 소스를 설치하는 것은 아무런 문제가 없다. 그러나 소스가 오리지널이 아니면, 소스를 빌드할 때 어떠한 것을 수정해야 하는지 참조할 만한 것이 없다. 결국 사용자는 원래 소스를 별도로 받아야 한다. RPM을 사용한다면, 여러분은 컴파일할 때 사용한 패치와 함께 원래 소스를 그대로 사용할 수 있다. 우리는 여기서 커다란 이득을 얻을 수 있다. 왜 일까? 여러 가지 이유가 있다. 하나는, 프로그램이 버전업되면, 여러분은 레드햇 리눅스에서 처음부터 컴파일할 필요가 없다. 그리고, 여러분은 어떠한 일을 할 필요가 있는지 보기 위하여 패치를 살펴볼 수 있다. 컴파일할 때 기본값은 이러 한 방법으로 쉽게 볼 수 있다.

RPM은 강력한 질의 옵션을 둘 수 있도록 설계되었다. 여러분은 전체 데이터베이스에서 특정한 패키지나 파일을 찾을 수 있다. 역시 여러분은 어떠한 파일이 어느 패키지에 담겨 있는지 쉽게 알아낼 수 있다. RPM 파일 자체는 압축되어 있지만, 알아야 필요가 있는 모든 (압축이 풀어진 형태의) 정보와 함께 패키지에 첨가한 특별한 바이너리 헤더 덕분에, 여러분은 개별적인 패키지를 *쉽고 빠르게* 검색 할 수 있다.

또하나의 뛰어난 기능은 패키지에 이상이 있는지 검증할 수 있는 능력이다. 걱정된다면 어떠한 패키지의 중요한 파일을 지우고, 검증해 본다. 여기서, 여러분은 필요한 패키지를 다시 설치할 수 있다. 가지고 있는 설정 파일은 모두 보존된다

우리는 RPM에 포함된 많은 아이디어와 개념을 제공한 BOGUS 제작진들께 감사하고 싶다. RPM은 전부 레드햇 소프트웨어사가 만든 반면에, 이러한 조작은 BOGUS의 코드에 기반을 둔다. (PM and PMS)

제 3 절 일반적인 정보

3.1 RPM 구하기

RPM을 구할 수 있는 가장 좋은 방법은 레드햇 리눅스를 설치하는 것이다. 만약 여러분이 그러기를 원치 않는다면, RPM만을 구하여 써 볼 수 있다. 이것은 *ftp.redhat.com* `<ftp://ftp.redhat.com/pub/redhat/code/rpm>` 에서 얻을 수 있다.

3.2 요구 사항

RPM을 사용하기 위해 주된 요구 사항은 cpio 버전 2.4.2 이상이 필요하다. RPM은 물론 리눅스에서 사용하고자 만들어졌지만, 다른 유닉스 시스템에도 포팅 할 수 있을 것이다. 사실은 SunOS, Solaris, AIX, Irix, AmigaOS, 그 외 여러 유닉스 에서 모두 컴파일된다. 다만 주의할 것이 있다면, 서로 다른 유닉스 시스템에서 만들어진 바이너리 패키지는 호환되지 않는다.

RPM을 설치하기 위한 최소 요구 사항은 다음과 같다. RPM을 소스에서 빌드하기 위해서, 여러분은 패키지를 컴파일하는데 필요한 것들, 즉, `gcc`, `make` 등이 필요하다.

제 4 절 RPM 사용하기

가장 간단한 형태로, RPM 은 패키지를 설치할 때 다음과 같이 사용할 수 있다:

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

다음의 간단한 명령은 패키지를 제거할 때 쓰는 것이다:

```
rpm -e foobar
```

매우 쓸모 있지만 더욱 복잡한 명령중 하나는 여러분이 FTP를 통하여 설치하는 것이다. 여러분이 네트웍에 연결되어 있고 새로운 패키지를 설치하기를 원한다면, 여러분에게 필요한 것은 파일의 정확한 URL과 함께 파일의 위치를 정하는 것인데, 다음과 같다:

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

이제는 FTP 를 통한 설치와 질의 기능을 사용할 수 있다. (ftp/bin 디렉토리에 rpm 바이너리를 가져다 놓기 바란다. 그렇게 한다면 당신의 ftp 서버는 rpm 질의를 받을 수 있다.)

여기의 간단한 명령중에, rpm의 사용 방법 메세지는 다양한 방법으로 사용이 가능하다는 것을 보여주고 있다:

```
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}
```

각 옵션에 대한 더 자세한 내용은 RPM 메뉴얼 페이지에서 볼 수 있다.

제 5 절 이제 우리가 RPM을 가지고 할 수 있는 것은 무엇이 있는가?

RPM은 매우 유용한 도구이고, 위에서 보듯이 다양한 옵션을 갖추고 있다. RPM을 이해하는 가장 좋은 방법은 몇 가지 예제를 살펴보는 것이다. 여기에서 간단한 설치/삭제를 포괄한, 많은 실례를 들었다:

- 실수로 몇 가지 파일들을 지웠는데, 어느 것을 지웠는지 확신할 수 없다. 여러분이 전체 시스템을 점검해 보고 어떠한 파일이 빠져 있는지 보려면, 다음과 같이 한다:

  ```
  rpm -Va
  ```

- 알 수 없는 파일을 보게 되었다. 어떠한 패키지 안에 있는지 보려면, 다음과 같이 한다:

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

  결과는 다음과 같다:

  ```
  xjewel-1.6-1
  ```

- 여러분은 kouls RPM을 발견하였지만, 이것이 무엇인지 알 수 없다. 이 파일에 대한 정보를 알고자 한다면, 다음과 같이 한다:

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

출력은 다음과 같다:

```
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.
```

- 이제 여러분이 kouls RPM을 설치할 때 어떠한 파일이 있는지 보려고 한다:

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

출력은 다음과 같다:

```
/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
```

지금까지 몇 가지 예를 들었다. 더 많은 것들은 RPM에 익숙하게 되면 쉽게 알수 있을 것이다.

제 6 절 RPM 만들기

RPM을 만드는 것은 무척 쉽다, 특히 여러분이 만들고자 하는 패키지를 얻었을 때는 더욱 그렇다. RPM을 만드는 기본적인 절차는 다음과 같다.

- `/etc/rpmrc`가 있는지 확인한다.
- RPM을 만들고자 하는 소스 코드를 구한다.
- 정확하게 빌드하기 위해서 소스에 필요한 패치를 가한다.

- 패키지에 대한 명세 파일을 만든다.
- 모든 것이 정확한 위치에 있는지 확인한다.
- RPM을 사용하여 패키지를 만든다.

보통, RPM은 바이너리와 소스 모두 만든다.

6.1 rpmrc 파일 (The rpmrc File)

RPM 설정 파일은 /etc/rpmrc 파일에서만 이루어진다. 예를 들면 다음과 같다:

```
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_vendor에는 RPM이 vender 줄을 찾을 때 필요하다. 이 줄은 /etc/rpmrc 명세 파일의 헤더에서 나온다. 이 기능을 끄려면, 숫자를 0으로 바꾼다. 같은 방법은 require_distribution과 require_group에서도 적용이 가능하다.

다음 줄은 distribution 줄이다. 여러분은 여기 또는 명세 파일 헤더의 뒷부분에 정의할 수 있다. 특정한 배포본에서 만들 때, 이 줄이 맞는지 확인하는 것은 매우 좋은 생각이다. 이것이 필요한 것은 아니지만, vender 줄도 같은 방법으로 이루어진다. 그렇지만 무엇이든지 올 수 있다. (예: Joe's Software and Rock Music Emporium).

RPM은 역시 다양한 아키텍처에서 패키지를 만드는 기능을 지원하고 있다. rpmrc 파일에는 아키텍처에 종속적인 플래그가 필요한 것을 빌드할 때 "optflags" 변수를 설정할 수 있다. 다음 단락에서 이러한 변수를 어떻게 사용하는지 볼 수 있다.

위에 있는 매크로에 더해서, 여기에는 몇 가지 더 있다. 여러분은 이렇게 사용할 수 있다:

```
rpm --showrc
```

태그가 어떻게 세팅되는지, 사용 가능한 플래그가 어떤 것이 있는지 알기 위해 서는 위와 같은 명령을 내린다.

6.2 명세 파일 (The Spec File)

우리는 명세 파일에 대해 논의할 것이다. 명세 파일은 패키지를 만드는데 필요하다. 명세 파일에는 소프트웨어와 설치할 모든 바이너리와 그 파일 리스트를 어떻게 만드는지에 대하여 지시한 설명이데, 이는 소프트웨어에 따라오는 것이다.

여러분은 명세 파일을 표준 관례에 따라 이름짓기를 원할 것이다. 명세 파일 이름은 이름-버전번호-발표 번호.spec이 된다.

여기에 간단한 명세 파일이 있다. (vim-3.0-1.spec):

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

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

%build
make RPM_OPT_FLAGS="$RPM_OPT_FLAGS"

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

%files
%doc README COPYING ChangeLog

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

6.3 헤더 (The Header)

헤더에는 여러분이 채워넣을 필요가 있는 표준적인 필드를 담고 있다. 여기에는 몇 가지 주의할 것이 있다. 필드에는 반드시 다음과 같이 채워야 한다:

- `Summary:` 패키지에 대한 간단한 설명을 한 줄로 쓴다.
- `Name:` 여러분이 사용하고자 하는 rpm 파일 이름이 와야 한다.
- `Version:` 여러분이 사용할 파일 이름의 버전 번호가 와야 한다.
- `Release:` 여기에는 같은 버전의 패키지 번호가 온다. (예를 들어 우리가 패 키지를 만들었는데 잘못된 것을 알고 다시 만들었을 때 다음 패키지의 release 번호는 2가 된다.)
- `Icon:` 여기에는 다른 (레드햇의 “glint;;와 같은) 시각적인 설치 도구 에서 사용할 아이콘 파일의 이름이 온다. 아이콘은 반드시 gif 포맷이어야 하고 `SOURCES` 디렉토리에 위치하여야 한다.
- `Source:` 이 줄에서는 원래 소스 파일의 위치를 가리킨다. 이것은 여러분이 소스 파일을 다시 얻거나 새로운 버전을 체크하는데 쓰인다. 주의: 이줄에서 파일 이름은 여러분의 시스템에 있는 파일 이름과 일치해야 한다. (예를 들어, 소스 파일을 다운로드 받아서 이름을 바꾸지 말아야 한다.) 여러분은 하나 이상의 소스 파일을 다음과 같은 라인을 사용하여 지정할 수 있다.

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

이 파일들은 SOURCE 디렉토리에 위치한다(이 디렉토리 구조는 뒤의 "소스 디렉토리 트리" 단락에서 다룰 것이다.)

- Patch: 패치를 찾을 수 있는 위치이다. 다시 다운로드 받을 때 필요하다. 주의: 여기서의 파일 이름은 "여러분의" 패치를 만들 때 사용하는 것과 일치하여야 한다. 여러 소스에서 여러 패치 파일을 가지고 있을 때 주목할 필요가 있다.

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

이 파일들은 SOURCES 디렉토리 안에 있어야 한다.

- Copyright: 이 줄에서는 패키지의 저작권을 알려준다. 여러분은 APL, BSD, MIT, 공개(public domain), distributable, 또는 상용 (commercial)과 같이 쓸 수 있다.
- BuildRoot: 이 줄에서는 새로운 패키지를 설치하고 만드는 "root" 디렉토리를 지정하도록 한다. 여러분은 설치하기 전에 여러분의 패키지를 테스트하는데 이를 사용할 수 있다.
- Group: 이 줄은 (레드햇 "glint"와 같은) 시각적 설치 프로그램에서 특정한 프로그램이 트리 구조에서 어디에 위치하는지 알려준다. 그룹 트리는 현재 이러한 구조를 가지고 있다.

```
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 이것은 헤더 아이템이 아니지만, 여기서 설명해 두어야 한다. 당신의 패키지, 서브 패키지마다 설명을 하나씩 필요로 할 것이다. 이것은 패키지에 대해서 참고적인 설명을 쓰는데 사용하는 것이고 여러 줄에 걸쳐 쓸 수 있다:

6.4 준비 (Prep)

이것은 명세 파일의 두번째 단락이다. 이것은 소스를 빌드할 준비를 하는데 쓰인다. 여기에서는 여러분이 소스 패치,make를 실행과 같은 셋업하는데 필요한 것들을 할 수 있다.

한가지 주의할 점: 각각 단락에는 실행할 쉘 스크립트가 위치하여야 한다. 여러분은 간단히 소스를 풀고 패치할 쉘스크립트를 만들어 %prep 뒤에 위치 시킬 수있다. 여기에서 우리는 도움이 되도록 매크로를 만들어 두었다.

매크로의 첫 번째는 %setup 매크로이다. 이것은 간단한 양식으로써 (명령행 옵션은 없다), 소스를 풀고 소스 디렉토리로 들어가는 것이다. 여기에는 다음과 같은 옵션이 있다:

- -n name 에서는 리스트된 이름에 빌드할 디렉토리의 이름을 정하는데, 기본값은 $NAME-$VERSION이다. 다른 가능성이 있는 $NAME, ${NAME}${VERSION} 또는 사용하는 tar 파일이 올 수도 있다. (명세 파일 안에 있는 "$" 변수는 실제 변수가 아니라는 점에 주의하기 바란다. 그것은 실제로 샘플 이름이 위치할 곳을 나타내는데 쓰인다. 여러분은 변수가 아닌 패키지의 실제 이름과 버전을 사용할 필요가 있다.)
- -c untar를 실행하기 전에 디렉토리를 만들고 그곳으로 이동하는 것이다.
- -b # 는 그 디렉토리로 이동하기 전에 소스#의 압축을 풀 것이다.(untar) (-c와 함께 사용할 수는 없다.) 이것은 여러 개의 소스 파일이 있을 때만 유용하다
- -a # 는 디렉토리로 이동한 후에 소스#의 압축을 풀 것이다.

- -T 옵션은 압축을 푸는 기본 기능을 무시하고 압축 풀린 소스 파일을 얻기 위하여 -b 0 또는 -a 0 를 필요로 한다. 부차적인 소스가 있을 때 이 옵션이 필요하다.
- -D -D 는 소스를 풀기 전에 디렉토리를 지우지 않는 옵션이다. 이것은 여러분 이 하나 이상의 셋업 매크로를 가지고 있을 때만 유용하다. 이것은 셋업 매크로 중 첫번째 것을 사용한 후에 쓰인다. (첫번째에 있으면 절대 안된다.)

사용 가능한 매크로중 다음으로는 %patch 매크로이다. 이 매크로는 소스에 패치를 가하는 과정을 자동화 하는 것을 돕는다. 여기에는 몇 가지 옵션이 있다. 다음과 같다:

- # 는 패치 파일로 패치 # 를 적용한다.
- -p #는 패치 명령(patch(1))을 strip할 디렉토리의 수를 지정한다.
- -P 의 기본 기능은 패치를 적용하는 것이다. 이 플래그는 기본 기능이고 압축 풀린 메인 소스 파일을 얻기 위해서 0이 하나 필요하다. 이 옵션은 첫 번째 매크로와 다른 숫자를 필요로 하는 두번째 %patch 매크로에서 유용하다.
- 여러분은 역시 실제 명령을 내리는 대신 %patch# 를 쓸 수 있다: %patch # -P

%build 매크로에서 여러분이 포함하고자 하는 모든 것(다음 단락에서 논의할 것 이다.)은 쉘을 통하여 실행하는 것이다. 여기서 여러분이 원하는 모든 형태의 매크로에 대해서는 예제를 보라.

6.5 빌드

이 단락에서는 실제로 어떤 매크로가 있는 것은 아니다. 여러분은 압축 풀린 소스를 가지고 있을 때 사용하기 원하는 소프트웨어를 빌드하고, 그것을 패치하고 그 디렉토리로 이동하는 등의 어떠한 명령이든지 여기에 넣어야 한다. 이것은 명령들의 쉘에 전달되는 또다른 셋으로써, 어떠한 쉘 명령이든지 (설명을 포함해서) 여기에 쓸 수 있다. 여러분의 현재 작업 디렉토리는 각각의 단락마다 소스 디렉토리의 최상위 레벨 디렉토리로 리셋되므로, 따라서 이것을 기억하기 바란다. 여러분은 필요하다면 서브 디렉토리로 이동할 수 있다.

6.6 설치

이것 역시 실제 어떠한 매크로가 아니다. 여러분은 기본적으로 설치하는데 필요한 어떠한 명령이든지 여기에 넣기를 원할 것이다. 여러분이 빌드하는 패키지 안에서 make install을 사용할 수 있다면, 여기에 넣어 두도록 한다. 아니면, 여러분은 make install에 쓰일 makefile을 패치하거나 make install을 여기서 할 수 있다, 또는 수동적인 쉘 명령으로 설치할 수도 있다. 여러분은 현재 디렉토리가 소스 디렉토리의 가장 상위 디렉토리가 된다는 것을 생각하여야 한다.

6.7 설치와 제거의 선행/후행 스크립트

여러분은 바이너리 패키지의 설치나 제거 전후에 스크립트를 넣을 수 있다. 주된 이유는 공유 라이브러리를 담고 있는 패키지를 설치하거나 제거하고 나서 ldconfig와 같은 명령을 실행하기 위해서이다. 각각의 스크립트에 대한 이 매크로들은 다음과 같다:

- %pre 설치하기 전에 실행되는 스크립트이다.
- %post 설치한 후에 실행되는 스크립트이다.
- %preun 제거하기 전에 실행되는 스크립트이다.
- %postun 제거한 후에 실행되는 스크립트이다.

이 단락의 내용은 #!/bin/sh 를 필요로 하지 않더라도 어떠한 쉘 스타일의 스크립트가 될 수 있다.

6.8 파일

여기는 여러분이 *반드시* 파일을 반드시 리스트해야 하는 단락이다. RPM은 `make install`의 결과로 어떠한 바이너리가 설치되는지 알 방법이 없다. 이것을 할 수 있는 방법은 "없다". 어떤 이는 패키지를 설치 전후에 find를 실행하기를 제의하기도 하였다. 다중 사용자 시스템에서는, 패키지 빌드가 이루어지는 동안 패키지 자체와는 아무런 관련이 없는 다른 파일이 생성될 수 있기 때문에 받아들이기 어렵다.

여기에는 특별한 작업에 사용 가능한 몇 가지 매크로가 있다. 여기에서 설명한다:

- `%doc` 여러분이 바이너리로 설치하기를 원하는 소스 패키지 내의 문서를 표시하는데 사용된다. 문서는 `/usr/doc/$NAME-$VERSION-$RELEASE`에 설치될 것이다. 여러분은 매크로를 써서 명령행에서 여러 문서를 리스트하거나, 모두 각각 매크로를 써서 리스트할 수도 있다.
- `%config` 는 패키지에서 설정 파일을 표시하는데 사용한다. 여기엔 sendmail.cf, passwd와 같은 파일을 포함한다. 여러분이 나중에 설정 파일을 담고 있는 패키지를 제거하고자 한다면, 수정하지 않은 파일은 모두 제거되고 수정된 파일은 `.rpmsave` 를 붙여 이름을 바꾸어 둔다. 여러분은 역시 이러한 매크로로 여러 개의 설정 파일을 리스트할 수 있다.
- `%dir` 파일 안의 패키지에 포함되는 파일 리스트 안의 단일 디렉토리를 표시한다. 기본값으로, 여러분은 디렉토리 이름을 `%dir` *없이* 나열할 수 있다, 디렉토리의 *모든것*은 파일 리스트 안에 포함되고 나중에 패키지의 한 부분으로 설치된다.
- `%files -f` <filename> 로는 소스의 빌드 디렉토리 안에 있는 몇몇 임의의 파일에서 여러분의 파일을 리스트할 수 있다. 이는 여러분이 파일 리스트를 직접 빌드할 수 있는 패키지를 가지고 있는 경우에 좋다. 여러분은 여기 있는 파일 리스트만 포함시키고, 여러분은 특별히 파일을 리스트할 필요가 없다.

파일 리스트에서 가장 주의해야 할 것은 디렉토리 리스트이다. 여러분이 실수로 `/usr/bin`을 써 두었다면, 여러분의 바이너리 패키지는 여러분 시스템의 `/usr/bin` 안의 모든 파일을 담게 될 것이다.

6.9 빌드하기

6.9.1 소스 디렉토리 트리

여러분에게 가장 필요한 것은 적절히 맞추어진 빌드 트리이다. 이것은 `/etc/rpmrc` 파일에서 설정 가능하다. 대부분의 사람들은 `/usr/src` 를 사용할 것이다.

여러분은 빌드 트리를 만들기 위해 다음과 같은 디렉토리를 생성할 필요가 있다:

- `BUILD` 는 RPM에 의해서 모든 빌드가 이루어지는 디렉토리이다. 여러분은 특정한 곳에서 빌드 테스트를 할 필요는 없지만, 이 디렉토리가 RPM이 빌드할 위치이다.
- `SOURCES` 오리지널 소스 tar 파일과 패치를 넣어 두어야 하는 디렉토리이다. 이는 기본적으로 RPM이 참고하는 곳이다.
- `SPECS` 은 명세 파일이 위치할 디렉토리이다.
- `RPMS` 는 RPM이 바이너리 RPM을 빌드할 디렉토리이다.
- `SRPMS` 모든 소스 패키지가 놓여질 곳이다.

6.9.2 빌드 테스트

아마도 여러분이 가장 원하는 것은 RPM 없이 깨끗하게 컴파일되는 소스를 구하는 것이다. 이를 위해서는, 소스를 풀고 $NAME.orig 디렉토리로 이동한다. 그리고 소스를 다시 푼다. 빌드에서 이 소스를 사용하도록 한다. 소스 디렉토리로 이동하고 빌드하기 위해서 명령을 따른다. 여러분이 편집해야 할 것이 있다면, 패치를 필요로 할 것이다. 빌드하고 나면, 소스 디렉토리를 지운다. 설정 스크립트에서 만들어진 모든 파일들이 지워지는지 확인한다. 그 다음 다시 소스 디렉토리로 이동한다. 그 다음 여러분은 다음과 같은 명령을 내린다:

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

이는 여러분이 명세 파일에서 사용할 수 있는 패치를 만드는 것이다. 여러분이 패치 파일 안에서 보는 "linux"는 동일한지 확인하기 위한 것(identifier)에 불과하다는 것에 주목한다. 여러분은 "config", "bugs"와 같은 패치를 만들어야만 하는 이유를 다룬 좀 더 자세한 설명과 같은 것을 원할는지 모르겠다. 바이너리가 실수로 포함되지 않는지 확인하기 위해서 사용하기 전에 패치 파일을 들여다 보는 것 역시 좋은 생각이다.

6.9.3 파일 리스트 생성

이제 여러분은 빌드할 소스를 가지고 있다. 그리고 빌드하고 설치하는 방법을 알고있다. 설치 과정의 출력을 보고 명세 파일 안에서 사용할 파일 리스트를 만든다. 우리는 일반적으로 명세 파일을 이러한 과정으로 동시에 만든다. 여러분은 초기화된 것을 만들고 쉬운 부분을 채울 수 있다. 그리고, 진행할 때 다른 곳을 채워 나간다.

6.9.4 RPM으로 패키지 만들기

여러분이 명세 파일을 갖게 되면, 여러분은 패키지를 빌드할 준비가 된 것이다. 가장 쓸만한 방법으로는 명령행에서 다음과 같은 명령을 내리는 것이다:

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

여기에는 유용한 `-b` 스위치와 함께 다른 옵션이 있다.

- `p` 는 명세 파일의 `prep` 단락을 실행한다는 것을 의미한다.
- `l` 은 리스트 체크이다.
- `c` 는 prep를 하고 컴파일한다. 이것은 여러분이 어떠한 소스를 빌드해야 할지 정확하지 않을 때 유용하다. 소스를 빌드하고 RPM을 사용하기 시작할 때까지는 여러분이 소스만 가지고 작업할지도 모르기 때문에 쓸모 없게 보인다. 그렇지만 RPM을 사용하는데 익숙해지면, 여러분은 이것을 사용할 때. 실례로써 찾을 수 있을 것이다.
- `i` 는 prep 컴파일, 설치를 한다.
- `b` 는 prep 컴파일, 설치와 바이너리 패키지만 만든다.
- `a` 는 소스와 바이너리 모두 만든다.

`-b` 스위치에는 몇 가지 수정 옵션이 있다. 다음과 같다:

- `--short-circuit` 은 특정한 단계를 바로 건너뛴다. (c와 i에서만 쓸 수 있다.)
- `--clean` 은 작업이 끝나면 빌드 트리를 지운다.
- `--keep-temps` /tmp에 만들어진 모든 임시 파일과 스크립트 을 그대로 둔다. 여러분은 `-v>` 옵션을 사용하여 실제로 tmp에 어떠한 파일이 만들어지는지 볼 수 있다.
- `--test` 는 실제 어떠한 단계도 실행하지 않는다, 다만 임시로 보존한다.

6.10 테스트

여러분이 소스와 바이너리의 rpm 패키지를 가지고 있으면, 시험해 볼 필요가 있다. 가장 좋은 방법은 여러분이 빌드한 것을 다른 머신에서 사용해 보는 것이다. 결국, 여러분은 `make install`을 여러분의 머신에서 여러번 해볼 것인데, 그것은 반드시 잘 설치되어 있어야 한다.

여러분은 패키지를 시험하기 위해 `rpm -u [패키지 이름]` 를 실행할 수 있다. 그러나 여러분은 패키지를 빌드할 때, `make install`을 하였기 때문에 속을 수 있다. 파일 리스트에서 어떤 파일을 빠뜨렸다면, 그것은 제거되지 않을 것이다. 여러분은 바이너리 패키지를 다시 설치하면 여러분의 시스템은 다시 완전해질 것이지만, rpm은 그렇지 않다. `rpm -ba [패키지 이름]` 을 실행시켰기 때문에, 대부분의 사람들은 `rpm -i [패키지]`로 설치한다는 것을 확인하고 명심하라. 바이너리가 홀로 설치될 때 여러분이 빌드하거나 설치할 때 수행되어야 할 필요가 있는 것중 하지 않은 일이 있는지 확인하라.

6.11 새로운 RPM 패키지들로 할 수 있는 것

어떤 RPM 패키지를 만들고 나면 (아직 RPM으로 만들어지지 않은 것으로 가정한다.) 여러분은 여러분이 작업한 것을 다른 사람들이 사용할 수 있도록 기여 할 수 있다. (역시 RPM으로 만든 것이 자유롭게 배포될 수 있다는 것을 가정한다.) 그렇게 하려면, 여러분은 *ftp.redhat.com* `<ftp://ftp.redhat.com>`에 업로드 할 수 있다.

6.12 지금은 무엇을?

테스트, 새로운 RPM 패키지들로 할 수 있는것" 단락을 보기 바란다. 우리는 구할 수 있는 모든 RPM을 사용할 수 있고, 우리는 그들에게 RPM이 도움이 되기를 원한다. 시험할 시간을 충분히 갖기를 바라고, 모든 이들이 그러한 혜택을 누릴 수 있도록 업로드하기를 바란다. 또한 여러분이 자유롭게 배포 가능한 소프트웨어만을 업로드 하기를 확인하기 바란다. 상용 소프트웨어와 쉐어웨어는 저작권에서 허가하지 않는한 업로드되어서는 안될 것이다. 이를 포함하고 있는 것은 Netscape software, ssh, pgp 등이 될 것이다.

제 7 절 다중 아키텍처에서 사용할 수 있는 RPM 만들기

RPM은 인텔 i386, 디지탈 알파 리눅스, 스팍용 패키지를 만드는데 사용할 수 있다. RPM은 SGI와 HP 웍스테이션에서도 잘 동작한다고 보고되었다.여기에는 패키지를 모든 플랫폼에서 쉽게 빌드할 수 있는 몇 가지 특징이 있다. 첫 번째 것으로는 `etc`rpmrc/의 "optflags" 지시자가 있다. 여기에서는 소프트웨어를 빌드할 때 아키텍처에 종속된 플래그를 세팅 할 수 있다. 명세 파일 안에 있는 다른 기능으로 "arch" 매크로가 있다. 그것은 여러분이 만드는 아키텍처에 의존되는 서로 다른 것들을 다루는데 사용할 수 있다. 또다른 기능으로 헤더의 "Exclude" 가 있다.

7.1 명세 파일 예제

여기에 나오는 것은 "fileutils" 패키지의 명세 파일의 일부분이다. 알파와 인텔에서 모두 빌드할 수 있도록 셋업하였다.

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

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

The ls program in this package now incorporates color ls!

%prep
%setup

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

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

.
.
.
```

7.2 Optflags

이 예제에는, 어떻게 etcrpmrc/에서 "optflags" 지시자가 쓰이는지 볼 수 있다. 빌드하고자 하는 아키텍처가 어떤것인지에 따라 다르지만, RPM_OPT_FLAGS에 적당한 값이 주어진다. 여러분이 사용하고자 하는 (-m486 and -O2와 같은) 지시자 안의 이 변수를 사용하기 위해서는 패키지의 Makefile을 패치하여야 한다. 여러분은 소스 패키지를 설치함으로 그리고 소스를 풀고 Makefile을 검토 하는데 무엇을 할 필요가 있는지 더 잘 알수 있다. 그다음 Makefil의 패치를 보고 어떠한 변화가 이루어졌는지 본다.

7.3 매크로

`%ifarch` 매크로는 여기 있는 것들 중 가장 중요하다. 보통 여러분이 둘 이상 아키텍처에 한하는 특정한 패치를 만들 필요가 있는 경우, 여러분은 RPM에서 한 아키텍처에만 패치를 적용할 수 있다.

다음의 예제에서, fileutils는 64비트 머신에 대한 패치를 가지고 있다. 분명히, 여기서는 알파에 대해서만 패치가 적용되어야 한다. 따라서 우리는 64비트 매크로와 같이 `%ifarch` 매크로를 추가한다.

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

여기서는 알파 외의 아키텍처에서 패치가 적용되지 않을 것이라고 확인한다.

7.4 패키지에서 제외되는 아키텍처

여러분이 모든 플랫폼의 소스 RPM들을 하나의 디렉토리에서 관리할 수 있다. 우리는 특정한 아키텍처에서 만들어지는"exclude" 패키지의 기능을 수행하였다. 따라서 여기에는 여러분이 다음과 같이 할 수 있다

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

그리고 바르게 만들어진 패키지가 만들어진다. 여러분이 한 응용프로그램을 특정한 플랫폼에서 이식한 적이 없다면, 여러분이 할 일은 소스 패키지의 명세 파일의 헤더에 다음과 같을 줄을 추가하여 주는 것이다.

```
ExcludeArch: axp
```

그리고 패키지를 여러분이 빌드하고자 하는 플랫폼에서 다시 빌드한다. 여러분은 인텔에서 빌드할 수 있는 소스 패키지를 가지고 있고 알파에서는 이 과정을 간단히 건너뛸 수 있다.

7.5 마무리

여러 아키텍처 사용할 패키지를 만들기 위해서 RPM을 사용하는 것은 보통 두 플랫폼에서 패키지를 각각 구하는 것보다 쉽다. 더 어려운 패키지를 빌드하는 경우에는 훨씬 쉽다, 항상 그렇지만, 가장 도움이 될 만한 것은 RPM을 빌드할 때 비슷한 소스 패키지를 살펴보는 것이다.

제 8 절 저작권

이 문서와 모든 내용은 저작권에 의하여 보호받는다. 이 문서의 내용이 그대로 보존되는 한 재배포가 허용된다. 바꿔말하면 여러분은 형식을 바꾸어 출력하거나 그대로 배포할 수 있다.