Package management using RPM

Ian Collier

July 1999

Introduction

RPM, or the 'Red Hat Package Manager', is a powerful tool allowing the management of software packages. Management tasks which it can perform include installing, verifying, removing, upgrading, building and printing information about packages. RPM is documented at the worldwide web site `http://www.rpm.org/` in two forms: a 'HOWTO' file containing a summary of its usage, and a book written in LaTeX called '*Maximum RPM.*' It is chiefly used for managing the software distributed with Red Hat Linux (which contains some six hundred RPM packages, though not all are installed at once), but it has been ported to several commercial UNIX systems including Solaris and IRIX.

Features of RPM

Contents of an RPM package

When a package is built, the compiled programs and associated files which must be installed are packaged up into a *binary* RPM; not only that, but the sources used to compile the programs are packaged into a *source* RPM. The source RPM contains:

- the pristine sources from which the package is built;
- any local modifications to the sources in the form of patch files;
- any local configuration files or other data, and
- a '*spec*' file containing the following information:
  - the name and version of the package together with a summary and a longer description of what the package is for;
  - the names of the source files from which the package is built and (usually) where the source files were obtained from;
  - complete instructions for building and installing the package, in a form which can be carried out automatically by the computer.

RPM takes the summary and description information from the spec file and includes them in the packages it builds so that the text can be printed out on request. Other information which RPM packages up with the files in a binary RPM includes:

- the date and time the package was built;

- the total size of all the files installed by the package;
- the name, location, ownership and permissions of each file in the package;
- for each file, a flag to indicate if it is a documentation file or a configuration file, and
- an MD5 checksum of each file.

This information can be listed for any RPM file or for any installed package. In addition, RPM can find out which package contains any particular file on the system. The MD5 checksums can be used to verify all the files in an installed package; either the checksums stored when the package was installed or those recorded in the original binary RPM can be used for this purpose.

If a file from a package fails verification then the flag saying whether it is a configuration file is printed out along with its details; this makes it easier to see whether the file might have been edited deliberately. In addition, files which are marked as configuration files and have been edited are not removed but instead renamed whenever a package is removed or upgraded.

Advantages of RPM

Possible advantages which may make RPM suitable for building packages as part of the OUCL Package System include the following.

- Information we normally request from package installers is contained in the spec file; in particular: the names and source locations of any source packages, the local modifications required, and the instructions on how to compile the package.
- The compilation information in the spec file must be correct, because the RPM files (both binary and source) will not be built unless the compilation and installation proccesses succeed.
- For most packages it is sufficient when porting from Sparc to Intel (or vice versa) merely to issue a 'rebuild' command and let the machine get on with building the appropriate binary package. If the package is portable enough then it may even compile on a different operating system without any changes needed.
- Because the package is designed to be built automatically, it is not necessary to keep the source tree lying around, thus saving disk space. Should the compiled source tree be required then it can be rebuilt by installing the source RPM and issuing a 'compile' command.
- Because the source tree is not usually kept, it is not necessary to resort to tricks such as using 'lndir' to create parallel build trees for different architectures.
- When the next version of a package arrives, the instructions contained in the previous spec file may well still apply so that the new package can be built without effort. If there are incompatibilities then the old instructions give an outline which can be the basis of an attempt to compile the new package.
- It may be possible to obtain some of the packages we require from the latest distribution of Red Hat Linux, in which case they will come with a 'free' spec file which may work with a minimum of editing. Of course, some editing will be required in order to move the files from their Red Hat locations (usually in `/usr/bin`) into locations within the Package System.

In addition, the fact that RPM keeps lists of files may make it easier to install and remove packages. This is not normally an issue, since we store the files of each package together in their own subdirectory of `/PACK`; however, it helps with the TEX system because this is made up of several smaller packages, and the files contributed by each of these packages are recorded so that at all times it is possible to see which package contributed each file. When it comes to updating one of these packages, RPM will automatically remove just the files from that package and replace them with those from the new package. Finally, the use of RPM to build the TEX system has meant that documentation for each package could automatically be extracted and used to build a comprehensive web page.

Disadvantages of RPM

It is almost certainly true that making an RPM package (for the first time, at least) is more difficult than simply installing the package in the traditional way. Some of the reasons why include the following.

- Even for packages which are trivial to make, there is a certain amount of 'syntax' which must be placed in the spec file to make it work.
- If there are any local modifications to be made (for instance, in configuration files and Makefiles) then it is not sufficient simply to edit files; the changes must be recorded in a patch file so that RPM can apply them afresh each time the package is compiled. (But that does mean that the changes which were made are listed in a more obvious way.)
- The whole package must be built from scratch in order for the RPM files to be made. If an error occurs during the build process then RPM cannot simply continue from where it left off when the error is fixed. This can be time consuming (but it does mean that the source RPM thus built is guaranteed to compile cleanly in future). Similarly, if it is desired to move or change one file in a package, or if a typographical error is found in the package description, then the whole package must be rebuilt to incorporate the change.
- During the 'make install' process it is usual for the files in a package to be installed in a temporary directory instead of in their final locations. This occasionally requires nontrivial changes to the package's Makefile.

Using RPM

Locations of files

RPM is currently installed in a temporary location within `/fs/packages/misc`. The main program for Sparc Solaris is contained in `/fs/packages/misc/rpm/bin`; this directory also contains '`rpm2cpio`' which can be used for extracting the contents of an RPM file without actually installing it. Typing '`rpm -ql rpm`' will show all the other files which accompany the program (including manual pages). Versions of RPM for other architectures can be found in `/fs/packages/misc/rpm-i386/bin` and `/fs/packages/misc/rpm-mips/bin`.

The file `etc/rpmrc` within the main RPM directory contains installation-specific options. The main purpose of this file is to define where source files, spec files and RPM files are stored: these are directed to subdirectories of `/fs/packages/rpm` (which is located on the same filestore as `/fs/packages/TarFiles` because it has a similar function). For example, whenever the spec file mentions a source file, RPM will look for this source file in `/fs/packages/rpm/SOURCES`. Essentially, all the subdirectories of `/fs/packages/rpm` are workspace areas except for the `RPMS` and `SRPMS` directories, which contain finished packages.

On Solaris the `etc/rpmrc` file defines the name of each binary RPM to end with the string '`-sol.rpm`', which serves to distinguish packages built on Solaris from those built on Linux (though, as it turns out, the machine architecture name for Intel Solaris is '`i86pc`' rather than '`i386`' as used on Linux). The remaining definitions fill in some operating system information which was left out of the main RPM configuration: Solaris 2.7 was not listed as compatible with binaries built on earlier versions, and the '`usparc`' and '`i86pc`' machine architectures were not mentioned in the main configuration.

Basic options

Full usage of the '`rpm`' command will not be described here because this command is fully documented in a manual page. However, basic descriptions of the main operations follow.

Install

The basic command to install a package is '`rpm -i`' followed by the name of the RPM file. To make the process slightly more pretty it is possible to use the flags '`-ivh`' instead of just '`-i`'. Before installing the package, RPM will verify that the package is suitable for the current operating system and architecture, that any required packages are present, and that there are no conflicts (such as two packages being incompatible or containing files with identical names). If the checks succeed then RPM will install the contents of the package in their final locations. The final locations of files in a binary RPM are those recorded in the package (and which can be listed with the '`rpm -qpl`' command). The files from a source RPM will be installed into the '`SOURCES`' workspace directory – except for the spec file, which will be installed into the '`SPECS`' directory. Installing an RPM will overwrite any file already present on the system, unless it is listed as a configuration file, in which case the file will be renamed and a warning displayed.

Using the flag '`-U`' instead of '`-i`' will cause RPM to upgrade a package, which is similar to installing it except that any already-installed package with the same name will be removed first.

Uninstall

The command '`rpm -e`' followed by a package name is used to erase packages from the system. Before removing the package, RPM will check that no other package depends on it. When a package is erased, any file listed as a configuration file which has been changed since it was installed is renamed instead of being removed.

Verify

For verifying packages the command '`rpm -V`' followed by a package name is used. This checks each file in the package against its MD5 checksum, and also verifies that any dependencies are satisfied. A line is printed out for each file which fails verification. If '`rpm -Vp`' is used then this can be followed by the name of an RPM file, and installed files will be checked against the checksums recorded in that file rather than those remembered by the system when the package was installed.

Query

Various kinds of information about packages can be printed out. The query command is introduced by '`rpm -q`' and followed by other options. By default, information about installed packages will be displayed, but by adding the '`-p`' flag it is possible to extract information from RPM files. Common queries are: '`-i`', which displays the description and general information about a package, and '`-l`' which lists the package's contents. Typing '`rpm -qpil *`' will display the information and contents from every package file in the current directory, while typing '`rpm -qail`' will display the same information about every installed package (the '`-a`' flag denotes all installed packages).

The '`rpm -qf`' command can be used to find out which package owns a particular file.

Building a package

The key to building a package is the spec file. In this section the spec file from the 'detex' package will be used as an example. For a full description of how to write a spec file please see '*Maximum RPM.*'

The header

```
Name: detex
Summary: Strip TeX commands from a .tex file
Version: 2.7
Release: 1
Source: ftp://ftp.tex.ac.uk/tex-archive/support/detex.tar.gz
Copyright: distributable
Group: Applications/Publishing/TeX
BuildRoot: /tmp/%{name}-%{version}-root
```

Most of the information recorded in the header is self-explanatory. The name and version number of the package are of course recorded, as is the release number, which reflects the number of changes which have been made to the spec file since this version of the package was first built (normally, changes are due to bugs uncovered after the package has been in use for a while). The summary is a short description of the package.

The `Source:` tag records the name of the source file used to build the package and where it was obtained from. When RPM comes to unpack the source it will take the last component of the path name (in this case, `detex.tar.gz`) and look for it in the `SOURCES` directory. If there had been more than one source file then there would have been tags named `Source1:`, `Source2:` and so on in addition to the normal `Source:` tag (which has the nominal number zero).

A package may also have a `Url:` tag which points to a web page associated with the package.

The `Copyright:` tag records information about the package's licence. Possible values include 'GPL', 'BSD' and 'public domain', but 'distributable' seems to be the default value when the package is allowed to be distributed but is not covered by a well-known licence.

The `Group:` tag divides packages into general groups according to their purpose. The RPM documentation contains a list of possible group names.

Finally, the `BuildRoot:` tag gives the name of a temporary directory which will be used when 'installing' the package. The constructions '`%{name}`' and '`%{version}`' are *macros.* A macro may be defined anywhere in the spec file, but RPM also pre-defines some macros. The two macros used here expand to the name and version number of the package, respectively. These are sometimes also defined explicitly in the spec file.

The description

```
%description
Detex reads files and removes all comments and TeX control sequences,
writing the result on the standard output so that it can be
spell-checked or processed in some other way.
```

The `%description` tag introduces the longer description of the package. The description text continues until the next tag in the spec file.

Preparation

```
%prep
%setup -q -n detex
```

The preparation section is where the package sources are unpacked. If there were any patches or local modifications to apply, this section is where it would be done. The commands in this section – as in most of the other sections – are ordinary shell commands mixed with macros. After the macros have been expanded the commands will be placed in a temporary file and executed as a shell script using `/bin/sh -e`. (If the shell terminates with non-zero exit status then RPM will issue an error message and abort.)

In this instance, the preparation section contains just one macro: the `%setup` macro. This macro extracts the main source tarfile. By default, the macro expects the tarfile to create a directory named after the package name and version (as in '`detex-2.7`'), so the '`-n detex`' flag is used to indicate that the directory will actually be called '`detex`'. The '`-q`' flag instructs the macro to perform the extraction quietly.

When this much of the spec file has been written, it is possible to run the preparation sequence using the command '`rpm -bp`'. The following is what happens.

```
$ rpm -bp /fs/packages/rpm/SPECS/detex.spec
Executing: %prep
+ umask 022
+ cd /fs/packages/rpm/BUILD
+ cd /fs/packages/rpm/BUILD
+ rm -rf detex
+ tar -xf -
+ /usr/local/bin/gzip -dc /fs/packages/rpm/SOURCES/detex.tar.gz
STATUS=0
+ [ 0 -ne 0 ]
+ cd detex
+ chmod -R a+rX,g-w,o-w .
+ exit 0
```

The sources have now been unpacked into `/fs/packages/rpm/BUILD/detex` and can be inspected.

Building

```
%build
CC=gcc make
```

Building the `detex` package is simple and requires only the '`make`' command. When the process is more complex it is sometimes a good idea to try the commands out as they are added to the spec file. When the `%build` section is complete it could be tried out with the command '`rpm -bc`'. Ordinarily this will also rerun the preparation section but this can be skipped by supplying the flag '`--short-circuit`.'

Installing

```
%install
rm -rf $RPM_BUILD_ROOT
pack=$RPM_BUILD_ROOT/PACK/%{name}/%{version}
bash -c "mkdir -p $pack/{bin,man/man1,pack,src,doc}"
cp -p detex $pack/bin
strip $pack/bin/detex
cp -p detex.1l $pack/man/man1/detex.1
if test -x /usr/bin/catman; then /usr/bin/catman -w -M $pack/man
elif test -x /usr/lib/makewhatis; then
   /usr/lib/makewhatis -M $pack/man $pack/man/whatis
fi
ls $pack/bin | \
   perl -lne 'print "/usr/bin/ln -s \$use_dir/bin/$_ \$use_home/packbin"' \
   > $pack/pack/use.sh
ls $pack/bin | \
   perl -lne 'print "/usr/bin/rm -f \$use_home/packbin/$_"' \
   > $pack/pack/unuse.sh
cp -p $pack/pack/use.sh $pack/pack/use.csh
cp -p $pack/pack/unuse.sh $pack/pack/unuse.csh
cp -p README $pack/doc
ln -s /fs/packages/rpm/SRPMS/%{name}-%{version}-%{release}.src.rpm $pack/src
```

The `%install` stage involves installing a complete mirror of all the package's contents in subdirectories of the 'build-root' directory. The build-root directory is the temporary directory named in the header of the spec file, and its name is copied to the environment variable `$RPM_BUILD_ROOT`. Each file will be installed in a location which mirrors its final location but which has this directory name prepended. (It is also possible to install without using a build-root directory, but this practice is discouraged.)

Sometimes, installing is as simple as setting the correct destination directory and typing '`make install`'. However, in this case simply copying the file is easier, and there is much more to be done besides in order to integrate the package into the OUCL Package System.

The first thing which is usually done is to make sure the build-root is clean and then create the appropriate subdirectories in it. The above script defines a shell variable `$pack` for ease of reference to the `/PACK/detex/2.7` directory within the build-root.

Next, the binary and manual pages are copied to their final locations, and the `windex` (for Solaris) or `whatis` (for IRIX) file is created.

The next few lines of the script are the equivalent of calling '`pvcsmakeuse`' to create the 'use' and 'unuse' files. At the moment that program is not configured to work in a build-root directory so these perl instructions are used instead.

The doc directory generally contains plain-text documentation which came with the package and doesn't belong anywhere else: in this instance the README file is copied there. (On Red Hat systems this documentation actually ends up in a subdirectory of /usr/doc and RPM has a macro to facilitate that, but it is not useful here.)

Finally, a symbolic link to the expected filename of the source RPM for this package is placed in the src directory.

This section of the spec file can be tested with the command 'rpm -bi.' Again, this will repeat the preparation and compilation stages unless the '--short-circuit' flag is added.

Clean-up

```
%clean
rm -rf $RPM_BUILD_ROOT
```

This section describes how to remove temporary files once the package has been successfully built. It almost always just contains a command to delete the build-root directory. Its position within the spec file varies.

File list

```
%files
%defattr(-,imc,support)
%docdir /PACK/%{name}/%{version}/doc
%docdir /PACK/%{name}/%{version}/man
/PACK/%{name}/%{version}
```

This section lists all the files which will be packaged up into the binary RPM. In this case there is only one file: /PACK/detex/2.7. Because this is a directory, RPM will treat it and everything below it as part of the package.

The %defattr macro gives the mode, owner and group which is to be recorded for all files (except for files which have these details given explicitly by the %attr macro). In most cases the mode is left unspecified (by writing a hyphen) so that the actual mode of the file will be recorded.

The %docdir macro names a directory which contains documentation. All files within the specified directory will be flagged as documentation.

There is also a %config macro which names a configuration file. The specified file will be added to the package and flagged as a configuration file. Because this macro adds the file to the package, care must be taken not to name the same file elsewhere in the file list, or RPM will complain that the file is listed twice.

Post-install script

```
%post
cd /PACK/%{name}
if test ! -d default; then ln -s %{version} default
else ln -s %{version} "new-`date +'%Y.%m.%d'`"
fi
prog=detex
if test ! -f /usr/local/bin/$prog; then
   ln -s /PACK/%{name}/default/bin/$prog /usr/local/bin || true; fi
```

```
if test ! -f /usr/local/man/man1/$prog.1; then
   ln -s /PACK/%{name}/default/man/man1/$prog.1 /usr/local/man/man1 || true; fi
```

There are several optional scripts which may be supplied in the spec file, of which two are used here. The first of these two is the post-install script. Whenever the package is installed, this script will be fed to `/bin/sh` immediately afterwards.

The purpose of this script is to ensure the appropriate symbolic links are set up. Firstly, whenever a package is installed it becomes either the 'default' version (if there is no default version already) or the 'new' version. The first four lines of the script implement this. Secondly, some programs have links placed in `/usr/local/bin` and `/usr/local/man` and the rest of the script tries to ensure this for the `detex` program. (Actually, this will not succeed unless the package is installed by the superuser, and there are other problems associated with doing that, so what this part of the script will usually do is print out some error messages which indicate which links need to be made. Adding '`|| true`' to the end of each line prevents RPM from complaining that the package could not be installed. The installer could then run '`rpm -q --scripts`' to print out the post-install script and find out which links need to be made.)

Post-uninstall script

```
%postun
cd /PACK/%{name}
if test ! -d default; then rm -f default
else for i in new*; do
   if test -h $i -a ! -d $i; then rm -f $i; ln -s default $i; fi
done; fi
prog=detex
if test ! -f /usr/local/bin/$prog; then rm -f /usr/local/bin/$prog || true; fi
if test ! -f /usr/local/man/man1/$prog.1; then
   rm -f /usr/local/man/man1/$prog.1 || true; fi
```

This script is run just after the package is removed, and does the opposite of the post-install script. If the default version of the package is removed then the symbolic link named '`default`' will be removed also; otherwise, any 'new' symbolic link which is no longer valid will be changed to point to the default version. After that, the symbolic links to the programs will be removed if they are no longer valid (and the last comment from the previous section also applies here).

Putting the package together

Typing '`rpm -ba`' followed by the name of the spec file will put the preparation, compilation and installation phases together. If there were no errors and all the files listed in the `%files` section were found then RPM will then write a source RPM and a binary RPM and execute the `%clean` section to remove the build-root directory. Adding the '`--clean`' flag to the command line causes RPM to remove the build tree as well. An example follows. Several lines have been omitted from the compilation and installation stages for the sake of brevity.

```
$ rpm -ba /fs/packages/rpm/SPECS/detex.spec
Executing: %prep
+ umask 022
+ cd /fs/packages/rpm/BUILD
+ cd /fs/packages/rpm/BUILD
+ rm -rf detex
+ tar -xf -
+ /usr/local/bin/gzip -dc /fs/packages/rpm/SOURCES/detex.tar.gz
STATUS=0
+ [ 0 -ne 0 ]
+ cd detex
+ chmod -R a+rX,g-w,o-w .
+ exit 0
Executing: %build
+ umask 022
+ cd /fs/packages/rpm/BUILD
+ cd detex
+ make
CC=gcc
...
gcc -O  -o detex detex.o -ll
+ exit 0
Executing: %install
+ umask 022
+ cd /fs/packages/rpm/BUILD
+ cd detex
+ rm -rf /tmp/detex-2.7-root
pack=/tmp/detex-2.7-root/PACK/detex/2.7
+ bash -c mkdir -p /tmp/detex-2.7-root/PACK/detex/2.7/{bin,man/man1,pack,src,doc}
+ cp -p detex /tmp/detex-2.7-root/PACK/detex/2.7/bin
...
+ exit 0
Processing files: detex
Finding provides...
Finding requires...
Prereqs: /bin/sh
Requires: libc.so.1 libdl.so.1
Wrote: /fs/packages/rpm/SRPMS/detex-2.7-1.src.rpm
Wrote: /fs/packages/rpm/RPMS/sparc/detex-2.7-1.sparc-sol.rpm
Executing: %clean
+ umask 022
+ cd /fs/packages/rpm/BUILD
+ cd detex
+ rm -rf /tmp/detex-2.7-root
+ exit 0
```

The binary RPM thus created can now be installed.

A feature worth noting from the above example is that RPM attempts to discover what facilities this package requires. If these are not present when the package is installed then RPM's dependency check will fail and RPM will not install the package (this can be overridden at install time with the '`--nodeps`' flag). The above example requires `/bin/sh` because it contains post-install and post-uninstall scripts which will require the shell. It also requires `libc.so.1` and `libdl.so.1` because these are dynamic libraries against which the `detex` executable is linked.

Porting the package

Unless there is anything particularly architecture-dependent in the package, a source RPM created on Sparc Solaris can usually be compiled immediately on Intel Solaris. Simply log in to an Intel workstation and type:

```
rpm --rebuild /fs/packages/rpm/SRPMS/detex-2.7-1.src.rpm
```

which will proceed to install the sources and then compile a binary RPM using the same steps as above. (Note that because RPM usually uses the same location for the build tree it is not a good idea to try to compile for multiple architectures at the same time). When the rebuild is complete, RPM will also remove the build tree, the source tarfile and the spec file. The latter can be recovered by re-installing the source RPM if necessary.

The `detex` RPM has also been compiled on IRIX using the same procedure.

The TeX system

Components

The TeX system is currently made up of about fifty packages, details of which can be seen on the LaTeX package web page.[1] Most of these are macro packages which can be installed or removed more or less independently, but there are a small number of essential packages which comprise the core of the system, including the TeX and LaTeX programs themselves. These core packages contain certain dependencies on each other and need to be built and installed in a particular order.

Order of installation

The `texklib` package contains a basic set of fonts, macros and support files which are required for building TeX and so it must be installed first. The build procedure just consists of unpacking the files and moving some of them around to suit local preferences. The LaTeX-related files are moved and bundled into a separate package; this package will not actually be required because LaTeX will be installed from fresh sources later.

The `texk` package is next to be built. This is the most complex package to build (and takes about ninety minutes) because it contains all the binaries and support scripts. After this is installed, the `texk-PACK` package is built (which gives TeX a directory within `/PACK`, thus turning it into a supported package). This is probably not absolutely necessary as long as the TeX binaries directory is placed into the `PATH` so that TeX can be used to build future packages.

Because LaTeX seems to use the AMS-fonts, the `amsfonts` package is built and installed next. This can be followed by the `amstex` package if desired.

Now everything necessary for LaTeX has been installed, the `latex` package itself can be built. This takes some time, mainly because DVI versions of all the source files are included. When this package is installed, RPM will print out a script which must be used if LaTeX is required before the `latex-PACK` package has been installed.

The `texk-misc` package should probably be built and installed next, because until this is done there are no PK fonts for the `xdvi` and `dvips` programs to use. Note that building this package

[1] at `http://www.comlab.ox.ac.uk/internal/PACK/latex/packages/index.html`

requires LaTeX. The main task carried out when the package is built is the creation of PK font files, and this takes several hours.

The rest of the packages can probably be installed in any order, with one or two exceptions. The main point to note is that building the `latex-PACK` package requires both `babel` and `xypic` because it includes those as subsidiary packages for convenience (notice that building this package creates three binary RPMs). Also, `cmps` must be installed before `bakoma` is built because the latter is designed to exclude all the fonts provided by the former when it is built; and `mdwfonts` requires `mdwtools` in order to format the documentation. It should also be noted that `tools`, `babel`, `graphics`, `amslatex`, `psnfss` and `cyrillic` are classed as 'required LaTeX packages' and so should be given priority.

Special considerations when installing TeX

There are a small number of peculiarities which make installing the TeX system slightly more difficult to install than most other packages.

Retrieving the tarfile

Most of the spec files for components in the TeX system contain lines such as the following indicating where the source tarfile came from.

```
Source: ftp://ftp.tex.ac.uk/tex-archive/macros/latex/base.tar.gz
```

Unfortunately, because of the way the TeX archive works this tarfile may not be retrieved simply by typing the above URL into a browser. Instead an ftp client must be used in a manner such as the following.

```
$ ftp ftp.tex.ac.uk
Connected to ftp.tex.ac.uk.
Name (ftp.tex.ac.uk:imc): ftp
331 Guest login ok, send your complete e-mail address as password.
Password:
230 Guest login ok, access restrictions apply.
ftp> cd /tex-archive/macros/latex/
ftp> bin
ftp> get base.tar.gz
```

The tarfile will be created on the fly, and changing directory before retrieving it will ensure that its contents will be placed in the correct directory when it is unpacked.

No-architecture RPM files

Most of the files in the TeX system are architecture-independent: macros, fonts, DVI files and so on. RPM has a special architecture called `noarch` for packaging such files. The main feature of a `noarch` binary RPM, apart from its name, is the fact that RPM will not complain if you compile the package on a Sparc workstation and then try to intall it on an Intel workstation. RPM does still expect the operating system to match, however (this can be suppressed by installing with the '`--ignoreos`' flag).

Building a package into a `noarch` RPM just requires the addition of an extra line in the spec file, as follows.

```
BuildArchitectures: noarch
```

Shared files

The fact that many of the files in the TEX system are architecture-independent made it desireable to use the non-shared `/PACK` directory only to store essential scripts and symbolic links; everything else has been installed in a central shared file system under `/fs/tex`. In fact, for various reasons even the compiled programs have been installed under `/fs/tex`, in separate directories for each architecture.

Unfortunately this creates a problem: installing a package on the Sparc architecture and then on the Intel architecture will cause most or all of the files to be installed twice. However, installing on multiple architectures is needed for the `texk` RPM because not all of its files are shared. Several other RPMs must also be installed on all the architectures in order to satisfy dependency checks (which are carried out during building as well as when installing).

Installing an identical `noarch` RPM twice should not cause any problems. RPM will simply replace each file with another copy of itself. If any of the files in the RPM were marked as configuration files, however, RPM will save a back-up of the original file and install a new copy. These back-up files will have to be found and returned to their original locations. RPM prints a warning for each relevant file whenever this happens.

Installing RPMs which share files but were built at different times (as will be the case with the `texk` RPM) causes a minor problem because some of the files will have different time stamps. In addition, some types of file including TEX format files and occasionally DVI files carry a time stamp within their content and so these files will also have different MD5 checksums. This means that when the second copy of the RPM is installed the first copy will contain several files which fail verification. In such cases the current situation is that the Sparc architecture is the one which should verify properly (in other words, the Sparc RPMs were installed last, even though in many cases they were built first).

Read-only filesystems

By default the `/fs/tex` filestore is mounted read-only on all machines (including the fileservers), and this is a problem when the packages come to be installed.

For installing the packages on the Intel and SGI systems there seems to be no option other than to make the auto-mounted filestore temporarily writable by typing '`umount`' and '`mount`' commands manually. This of course requires the server to share the appropriate filestore with the relevant host with write access.

A different approach has been taken for installing the packages on the Sparc architecture: files which belong in `/fs/tex` are actually installed directly in `/tex` on the fileserver (`gamma.comlab`), which is the writeable directory from which `/fs/tex` is served. This uses RPM's 'prefix' mechanism for relocatable files.

If all the files within a package share a common prefix (in the case of TEX-related files the prefix is `/fs/tex/tex7.2b` except in the case of LATEX, whose prefix is `/fs/tex/latex19991201`) then RPM allows this prefix to be specified in the header, as in the following extract from the spec file for the `latex` package.

```
%define prefix /fs/tex/latex%{version}
Prefix: %{prefix}
```

(The prefix is defined as a macro so that it can be referred to later in the spec file; this makes it easy to change the intended locations of the files in the package. The name of the prefix also includes the macro which will expand to the LATEX version number.)

When an RPM with such a prefix is installed, it is possible to specify a substitute prefix using the '`--prefix=`' flag. So, to install the `latex` package the following command would be used.

```
rpm -ivh --prefix=/tex/latex19991201 latex-19991201-1.noarch-sol.rpm
```

This replaces the prefix specified in the spec file with the prefix given on the command line. The replacement applies to all the files in the package (it is an error to include a file in the package which does not start with the named prefix).

The 'prefix' mechanism usually denotes that a package is relocatable; that is, the package will still work wherever it is intsalled. This is not strictly true with the TEX system: in particular, the `tex-PACK` and `latex-PACK` packages contain links and scripts which expect certain files to be installed in particular locations, and the `texk` package contains some scripts and configuration files which mention some absolute locations of files. That having been said, the architecture-independent tree could in theory be located anywhere provided the programs could be told where to find it.

Each of the packages which contain architecture-independent files within `/fs/tex/tex7.2b` contains scripts which call the '`mktexlsr`' command to update the filename database whenever the package is installed or removed. (The exception is the `texklib` package, which is meant to be installed before the `mktexlsr` command exists). In order for this command to work when `/fs/tex` is read-only, the `TEXMF` environment variable must be set to the root of the writeable TEX tree, namely `/tex/tex7.2b/texmf`.

Unfortunately, installing packages this way has a disadvantage: the database of file names will now have all files listed under `/tex` instead of under `/fs/tex`. This fact needs to be remembered whenever the contents of a package are listed, or when '`rpm -qf`' is used to find out which RPM contains a given file. In addition, package verification will only work on the file server where `/tex` is a valid directory.

Some of the packages which install into directories other than `/fs/tex` have also been given prefixes, but it is not in general necessary to relocate these packages.