Building RPMs

Bob Dowling  
rjd4@cam.ac.uk  
University of Cambridge  
Computing Service

This is the University Computing Service’s course on building RPMs, the software packages used by most Linux distributions. RPM stands for Red Hat Package Manager and was written by the creators of the Red Hat distribution. However, it is used much more widely than that.

This particular version of the course is based around the SuSE distribution. I will highlight any areas where the distributions differ significantly and attempt to demonstrate a fairly “distribution neutral” style of building packages.
Prerequisites

- Use of rpm with existing packages:
  - rpm --install ...
  - rpm --upgrade ...
  - rpm --freshen ...
  - rpm --query --info ...
  - rpm --verify ...
  - rpm --erase ...
- Using the command line.
- Using emacs or vi.

There are some serious prerequisites for this course. We need you to be familiar with a plain text editor (such as emacs or vi) and running commands from the command line (there's no GUI work in this course). We also need you to be familiar with using rpm on existing packages.
Outline of course

- Principles
- File layout
- Application worked examples
  - XiaoS
  - Sun's Java 2 SDK
- System daemon worked example
  - Java 2 ORB daemon (later course)

This course will start with a brief description of the principles underlying the RPM model. It will then cover the file layout used for building packages and demonstrate how to do all the work in your own home directory. While you need to be root to install or erase packages you don't ever need to be root to create them!

The vast majority of the course will be two worked examples. The first will be a single application called the “XiaoS” which is a real-time fractal program. The second will be Sun's Java 2 software development kit (SDK) which will include a system daemon, the orb.
Principles

- Build and installation scripted
- “Pristine” sources
- Package dependencies
- Pre-/Post-install/upgrade/uninstall scripts

Firstly the RPM philosophy says that we must script the building and installation of the software. This means that we get a concrete record of everything that needs to be done to set up the software starting with just the source code.

The source code itself is kept in exactly the state that the original software author created it. A distinction is drawn between the software author and the software packager (ourselves in this case). Any changes that need to be made to the source code so that it can work on our system are kept separate from the original source code which is kept in a “pristine state”. A side effect of this is that there are two version numbers, one for the original software and a second for our packaging of it.

We will also use the packaging system to explicitly label dependencies between packages. So if there is software that our package needs to work correctly we will identify the packages needed to provide that software.

As well as scripting the build and installation procedure, we can also provide scripts to be run at installation, upgrade and ensure time. This is intended to keep the amount of manual effort on the part of the system administrator to an absolute minimum.
Default working location

- **Red Hat**: /usr/src/redhat
- **SuSE**: /usr/src/packages
- **Custom**: wherever
- **Directory structure under that directory**:

```
/usr/src/redhat/
```
```
BUILD  RPMS  SOURCES  SPECS  SRPMS

i386  i686  noarch
```

On SuSE, by default rpm uses the /usr/src/packages directory tree. On a Red Hat system the corresponding /usr/src/redhat directory is only writable by root so is no use to us if we don’t have the root password, but on a SuSE system the directory hierarchy is world-writable in the same as /tmp. We will not have to be root to build packages just to install them. We will use a directory in /tmp to keep our work separated.

The directory tree under /usr/src/redhat or whatever is required by rpm. We will have to duplicate it wherever we want to build our packages.

**SOURCES**: This directory is where the source code, patches and extra files we add to packages are kept prior to their unpacking. Nothing that the package creating process does should ever update any file in this directory.

**BUILD**: This directory is where the software source code is unpacked and built. A subdirectory is created specifically for each package to keep them from clashing.

**SPECS**: This directory contains the control files that contain all the information for building the packages. There is at most one “spec file” per package. (Some spec files build more than one package.)

**RPMS**: This directory contains subdirectories for each of the architectures that can be built for. i386 is the “generic Intel” architecture. i686 is used for packages built for the Pentium 2 chip specifically. noarch is for packages that contain no architecture-specific binaries at all.

**SRPMS**: This directory contains the “source packages”. These files are a bundle of the spec file and the source files and make for a quick and easy way to rebuild the packages.
Practical: A custom directory

- We will use /tmp/userid
- Override rpmbuild's defaults
  - ~/.rpmmacros
  - %_topdir /tmp/userid
- PWF-specific command to populate the directory:
  ${UX}/Lessons/rpmbuild/setup

We will use a subdirectory of /tmp. For rpm to use /tmp/userid rather than /usr/src/packages we set one of its parameters in a per-user configuration file called ~/.rpmmacros. In that file we set a parameter, _topdir, to the value /tmp/userid. This directory must contain the needed directory tree. For the purposes of this course, a helper script has been written which will do this first.

- Make sure you do not currently have a ~/.rpmmacros file or a /tmp/userid directory.
- Run the command ${UX}/Lessons/rpmbuild/setup
  (That dollar is real and is to expand the UX variable. It is not a representation of the prompt.)
- Observe that you now do have a ~/.rpmmacros file and a /tmp/userid directory.

We are now ready to proceed.
Example application: XiaoS

- A complex application
  - But just an application
  - Fractal drawing program
- Documentation
- Configuration files
- etc.

During the first session of this course I am going to be working with an application called XiaoS. Now, this is a single application launched by the command `xaos`. I shall be working on this application to show how things work. During the practical sessions you will be trying out these techniques on another application: `nano`, a simple editor.
Getting started — the spec file

- The product: XaoS
- The version: 3.1
- Our package name: XaoS
- SPECS/XaoS.spec
- emacs creates a template on Red Hat
- We will copy dummy.spec

We start by setting up the control file known as the package’s "spec file" because it contains the specification of the package. This file needs to go in the SPECS directory and traditionally has a name given by the name of the package with the suffix .spec.

If you follow this naming convention on a Red Hat system the emacs editor will automatically set itself up for editing a spec file and if it is creating a file from scratch it will provide you with a very useful template whose package name is derived from the name of the file. The dummy .spec file is a copy of this template which we will use because SuSE doesn’t include this option in its release of emacs.
Macros

• Package macros `%{variable}
  - `%{name}, `%{version}, `%{release}
• System-wide macros `%{variable}
  - System file layout conventions via macros.
  - Especially useful for relocatable packages.
• rpm --eval '%{variable}'

We can see from the spec file that rpmbuild has a macro system where values can be assigned and recovered with the `%{variable}` syntax. The values of certain macros are specified by the Name:, Version:, and Release: header lines. The macros set up are `%{name}`, `%{version}`, and `%{release}`. We will meet (many) others later.

The system defines some global macros common overall packages. These might define where things go according to local policy and typically have names starting with an underscore, so they are `%{variable}` etc.

If you want to know the value of any system variable, rpm has an “--eval” option to evaluate a spec file fragment:

```
$ rpm --eval '%{bindir}'
/usr/bin
```

<table>
<thead>
<tr>
<th>macro</th>
<th>redhatvalue</th>
<th>purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>%{prefix}</td>
<td>/usr</td>
<td>User programs</td>
</tr>
<tr>
<td>%{bindir}</td>
<td>/usr/bin</td>
<td>Library include files</td>
</tr>
<tr>
<td>%{includedir}</td>
<td>/usr/include</td>
<td>Application libraries</td>
</tr>
<tr>
<td>%{libdir}</td>
<td>/usr/lib</td>
<td>System/administrator programs</td>
</tr>
<tr>
<td>%{sbin}</td>
<td>/usr/sbin</td>
<td>Support programs (not used directly)</td>
</tr>
<tr>
<td>%{libexecdir}</td>
<td>/usr/libexec</td>
<td>Support data files for programs</td>
</tr>
<tr>
<td>%{datadir}</td>
<td>/usr/share</td>
<td>Application configuration files</td>
</tr>
<tr>
<td>%{sysconfdir}</td>
<td>/etc</td>
<td>Files that record the state of programs</td>
</tr>
<tr>
<td>%{localstatedir}</td>
<td>/var</td>
<td>Manual pages</td>
</tr>
<tr>
<td>%{mandir}</td>
<td>/usr/share/man</td>
<td>Information pages</td>
</tr>
<tr>
<td>%{infodir}</td>
<td>/usr/share/info</td>
<td>Startup/shutdown scripts</td>
</tr>
<tr>
<td>%{initrd}</td>
<td>/etc/rc.d/init.d</td>
<td>Documentation files</td>
</tr>
<tr>
<td>%{defaultdocdir}</td>
<td>/usr/share/doc</td>
<td>Files that vary dramatically in size (e.g. log files)</td>
</tr>
</tbody>
</table>
Filling in the blanks

- **Summary:**
- **Name:**
- **Version:**
- **License:**
- **Group:**
- **URL:**
- **%description**

Now that we have the template we need to fill it in. There are a number of blank fields which are compulsory so we must fill in values:

- **Summary:** a *one line* description of the package
- **Name:** the name of the package
- **Version:** the version of the software
- **License:** The licence or copyright covering the software
- **Group:** The category of software. See the list below.
- **URL:** An address of a web page for this software

The `%description` statement is not an evaluated macro but a spec file statement that introduces a multiline description of the package.

All of these entries appear in the output of `rpm --info --query`:

```
$ rpm --query --info coreutils
Name : coreutils Relocations: (not relocateable)
Version : 5.0 Vendor: SuSE Linux AG
Release : 90 Build Date: 21 Oct 2003 17:19
Install Date: 25 Jun 2004 14:17 Build Host: amsim9.suse.de
Group : System/Base Source RPM: coreutils-5.0-90.src.rpm
Size : 6188096 License: GPL
URL : ftp://alpha.gnu.org/gnu/coreutils/
Summary : The GNU core utilities

Description :
*Basic file, shell and text manipulation utilities. The package contains the following programs:* ...
```
Software groups

- Differs between distributions
- Red Hat Linux 9
  - 29 groups
  - Two levels
- SuSE 9.0
  - 224 groups
  - Four levels

I'll concentrate on the groups offered by the two main RPM-based distributions: Red Hat and SuSE.

If you want to get the list of groups installed on your system, to get a feel for the nature of them on the particular distribution you use, try this:

```bash
$ rpm --query --queryformat '%{group}\n' --all | sort --unique
```

This command may take some time and you might want to redirect the output to a file.

The official set of group names can be found in the GROUPS file in the rpm package documentation. (On a Red Hat or Fedora system this is `/usr/share/doc/rpm-version/GROUPS` and on a SuSE system the file can be found in `/usr/share/doc/packages/rpm/GROUPS`. Interestingly, although SuSE publish this file they don't use it! They have a completely different set of groups. I will stick with the "official" group names.)
Red Hat Linux groups

• Six top-level categories:
  – Amusements, Applications, Development, Documentation, System Environment, User Interface
• Single level subcategories
  – e.g. Applications/Editors, User Interface/X

Red Hat has a simple, if rather unstructured approach. There are 29 groups in total:

Amusements/Games
Amusements/Graphics
Applications/Archiving
Applications/Communications
Applications/CPAN
Applications/Databases
Applications/Editors
Applications/Engineering
Applications/File
Applications/Internet
Applications/Multimedia
Applications/Productivity
Applications/Publishing
Applications/System
Applications/Text
Development/Debuggers
Development/Languages
Development/Libraries
Development/System
Development/Tools
Documentation
System Environment/Base
System Environment/Daemons
System Environment/Kernel
System Environment/Libraries
System Environment/Shells
User Interface/Desktops
User Interface/X
User Interface/X Hardware Support
SuSE groups

• Seven top-level categories:
  - Amusements, Development, Documentation, Hardware, Productivity, System, X11

• More complex structure
  - e.g. Amusements/Games/Strategy/Real Time, Productivity/Scientific/Chemistry

Here is a list of the top two levels of the SuSE categorisation.

Amusements
  Games, Toys
Development
  Languages, Libraries, Sources, Tools
Documentation
  HTML, Man, Other, SuSE
Hardware
  Fax, ISDN, Joystick, Mobile, Modem, Other, Palm, Printing, Psion, Radio, Scanner, TV, UPS
Productivity
  Archiving, Clustering, Databases, Editors, File utilities, Graphics, Hamradio, Multimedia,
  Networking, Office, Other, Publishing, Scientific, Security, Telephony, Text
System
  Base, Benchmark, Boot, Console, Daemons, Emulators, Fhs, Filesystems, GUI, I18n, Kernel,
  Libraries, Management, Monitoring, Packages, Shells, Sound Daemons, X11, YaST
X11
  Amusements, Applications, Games, Utilities
Changing the defaults

- Source code archive:
  - default: %name-%version.tar.gz
- Source unpacks to Xaos-3.1
  - %name-%version
  - RPM's default
- Change %changelog email address
  - Add line to .emacs file
  - (setq user-mail-address "rjd4@cam.ac.uk")

.emacs hazards a guess as to what the name of the source code file is. rpmbuild itself has no default. emacs follows a common convention. We will keep our source file name %name-%version.tar.gz but with a twist. We will actually set it to a URL.
Source: http://prdownloads.sourceforge.net/xaos/%name-%version.tar.gz
This does not mean that rpmbuild will do an automatic fetch of that URL. rpmbuild only looks at the last component of the URL (%name-%version.tar.gz) and expects to find a file of that name in the SOURCES directory. The advantage of using the full URL is that it gives a hint to future package maintainers on where they might find later versions of the source code.

rpmbuild assumes that the archive, whatever name it has, will unpack into a directory with the name %name-%version. In our case this is true. Special action is needed if it is not. See the next slide's notes for details.
Unpacking the source files

- `rpmbuild -bp XiaoS.spec`
- "-bp": Prepare
- Checks the syntax of the spec file
- Runs the %prep section
- Default: `%setup -q`
  - Finds source file in `.../SOURCES`
  - Unpacks into `.../BUILD`

The command to build an rpm package used to be `rpm` itself which took the `-b` or `--build` option to tell it to work in `build` mode. For reasons passing my meagre understanding Red Hat changed this to require the use of the `rpmbuild` command, all of whose useful options start with `-b`. After the "-b" comes a suboption identifying which stages of the package building process should be invoked. "-bp" runs the preparation phase which unpacks and reads the source code for compiling in a later phase.

What the preparation phase does is governed by the `spec` file between the `%prep` statement and the `%build` statement (which introduces the next phase). This is the core of a shell script (called a "scriptlet") which gets preceded by a set of environment variable definitions. It uses a macro, `%setup`, which expands into a series of commands that takes the source archive named in the `Source` header, finds it in the SOURCES directory, and unpacks it in the BUILD directory. The "-q" (for "quiet") option causes the untarring not to list every unpacked file.

If the archive unpacks into a directory called anything other than `%{name}-%{version}` then this name must be passed as the "-n" (for "name") option to `%setup` so if XiaoS-3.1.tar.gz unpacked into a directory `xaos31` rather than XiaoS-3.1 then we would use the command

```
%setup -q -n xaos31
```

Another possibility is that it does not create a directory at all but unpacks all over the current working directory. In this case we must instruct `%setup` to create the directory (which might as well be called `%{name}-%{version}`), change directory into it and only then unpack the archive. The option on `%setup` to trigger this behaviour is "-e" (for "create”).

In our case the `%prep` section is very simple:

```
%prep
%setup -q
```
Reading the instructions

• Once unpacked, take a look.
• File called “INSTALL” (doc/INSTALL)
• Instructions
  – configure
  – make
  – make install
• These will form %build and %install

Once we have unpacked the source archive we should look into it for instructions. These can typically be found in files called INSTALL or README. In our case, NagS has a file called INSTALL in the doc directory which instructs us to run three commands: “configure”, “make”, and “make install”. The first two commands form the building phase (the %build scriptlet) and the third the installation phase (the %install scriptlet). This triplet of instructions is so common that there is special support for it in rpmbuild.

NB: If you are writing the source code yourself there is no obligation to do it this way. configure, in particular, can be quite tricky to set up for, but make is well worth using. If you do choose not to use configure, please do provide some other mechanism to use the macros’ values.
Building

- `rpmbuild -bc XiaoS.spec`
- “-bc”: Compile
- Runs a fresh “prep” first
- The template's %build is empty
- Need to run
  - configure
  - make

At the start our %build section is empty. To trigger `rpmbuild` to run this phase we use the “-bc” option where the “c” stands for “compile”. Even with an empty %build section we see that the %prep stage is run first, guaranteeing a fresh start.

We need to fill in the %build section to run configure and make in order.
**configure**

- Sets up the build parameters
- Converts thing.in to thing
  - Makefile.in
- Program-specific parameters
  - ./configure --help
- System-specific parameters
  - %{configure} macro

configure has a single purpose: it takes various parameters passed to it on the command line or set to default values and writes various configuration files (Makefile, config.h, etc.) to take regard of their values. It generates these files from input files (Makefile.in, config.h.in, etc.) which have placeholders indicating which parameters should be substituted in.

To see if there are any settings to be made run "./configure --help" and look at the stream of options that pass by. Note that very many of the options (e.g. --sysconfdir) set system parameters whose values we have macros for. To set all of these, together with some compiler options, `rpmbuild` provides us with a `%{configure} macro that expands to a whole bunch of settings:

```
./configure
   --prefix=%{_prefix}       --exec-prefix=%{_exec_prefix}
   --bindir=%{_bindir}       --sbdir=%{_sbdir}
   --sysconfdir=%{_sysconfdir} --datadir=%{_datadir}
   --includedir=%{_includedir} --localstatedir=%{_localstatedir}
   --libdir=%{_libdir}        --libexecdir=%{_libexecdir}
   --sharedstatedir=%{_sharedstatedir} --mandir=%{_mandir}
   --infodir=%{_infodir}
```

Note that we can combine the macro with manual settings of program-specific settings:

```
%{configure} --enable-wombats=yes --enable-termite=no
```

In the specific example of NaoS it will be helpful to set:

```
%{configure} --with-mitshum=no --enable-nls=no
```
make

- %make?
  - Not in Red Hat — relies on configure
  - Present in SuSE
- make succeeds
- Actually there is an error lurking
  - We need to add a line to Makefile.in

There is no corresponding %{make} macro in Red Hat Linux. SuSE does offer one but Red Hat relies on %{configure} having set up everything necessary in the Makefiles. So we run plain make. This gives us a %build section looking like this:

%build
%{configure}
make

In our case everything seems to work well. But actually there's a problem lurking. Deep in the bowels of the Makefile are these three lines:

/usr/bin/install -c -d $(DESTDIR)$(mandir)/man6
/usr/bin/install -c -m 444 doc/xaos.6 $(DESTDIR)$(mandir)/man6
/usr/bin/install -c -m 444 doc/xaos.info $(DESTDIR)$(infodir)

Have you spotted the error? The directory $(DESTDIR)$(mandir)/man6 is created prior to having doc/xaos.6 installed in it. The corresponding "infodir" directory is not and it needs to be. Recall that Makefile is derived from Makefile.in by configure. We need to edit Makefile.in to add the necessary line.
Creating patches

• Need to change the source
• Want to keep source “pristine”
• Create a separate “patch file”
• SOURCES directory
• gendiff tool

We now have a dilemma. We need to change the source code but we want to leave it untouched. The way to deal with this is to create a distinct “patch file” which is a set of instructions to the patch command telling it what to change in the unpacked source tree. This lets us keep track of what the original developer did and what changes we, the packagers, have made.

This patch file will also go in the SOURCES directory. The package containing rpmbuild also provides us with a tool called gendiff (“generate differences”) to help us create this patch file.
Making a patch with gendiff — 1

- Start freshly prepped
  - `rpmbuild -bp XaoS.spec`
- Make "original" copies of file(s) to change
  - Use a common suffix (e.g. "rjd")
  - `cp Makefile.in makefile.in.rjd`
- Modify the unsuffixed file(s)
  - `Edit Makefile.in`

We always start from a freshly unpacked build tree. We will be applying the patch at the end of the %prep section so we will always generate it in the corresponding point in the process, before any of the %build has begun.

We change directory into the build tree and make copies of all of the files we are going to change. These copies must all be in the same directory as the files they are copies of and all have the same suffix appended. In this case, because I am fixing a bug I choose the .rjd suffix. Then we modify the file without the suffix so in our case we edit Makefile.in and leave Makefile.in.rjd etc. unedited.

```
$ cd ./BUILD/XaoS-3.1
$ cp Makefile.in Makefile.in.rjd
$ vi +38 Makefile.in
```

While only a single file is being changed here, multiple files could be changed (with their originals sharing a common suffix) with no change to the process.
Making a patch with gendiff — 2

- Run gendiff
- Be in BUILD/XaoS-3.1
- Put output in SOURCES directory
- `gendiff . .rjd > ../../
  SOURCES/XaoS-rjd.patch`
- `%{name}-suffix.patch`

To run gendiff we need to be at the top of the build tree (BUILD/XaoS-3.1 in our case). It takes two arguments: the directory tree it has to work with ("." in our case because we are in the relevant directory) and the suffix added to the copies of the original, unedited files (".rjd" in our case — including the leading full stop). It generates the patch file on standard output which should be redirected to a file in the SOURCES directory. A single patch file is always created even if multiple files have been changed. My preferred naming scheme for these patch files is `package-suffix.patch` (so the file is `XaoS-rjd.patch` in our case).

```
$ gendiff . .rjd > ../../SOURCES/XaoS-rjd.patch
$ cd ../../SOURCES
```
Applying the patch

- In the headers:
  Source0: ${name}-${version}.tar.gz
  Patch1: %{name}-rjd.patch

- In the %prep section:
  %prep
  %setup -q
  %patch1

Now we have our pristine source file (KaoS-3.1.tar.gz) and our patch file (KaoS-rjd.patch) that modifies the unpacked sources. We still need to apply the patch.

First we declare it. PatchN: lines follow the SourceM: lines.

The actual application of the patch during the %prep phase is done by the %patch command. %patchN applies the patch declared by the PatchN: line above:

  %prep
  %patch
  %patch1
Successful build

- Patching files
  - gendiff: .suffix
  - PatchN: %{name}-suffix.patch
- Macros
  - Let %{configure} and make do their thing

We have done everything needed to build a working system.

%prep
%setup -q
%patch1

%build
%{configure} --with-mitshm=no --enable-nls=no
make

Now it's your turn.
Practical — check setup

- `~/.rpmmacos` file
- `/tmp/ userid/... directory tree`
- `.../SOURCES/nano-1.2.3.tar.gz`

```
$ ls -l ~/.rpmmacos /tmp/rjd4
-rw-rw-r-- 1 rjd4 pwfread 21 Feb 4 12:47 /home/rjd4/.rpmmacos
/tmp/rjd4:
total 0
drwxrwxr-x 2 rjd4 rjd4 40 Feb 4 12:33 BUILD
drwxrwxr-x 8 rjd4 rjd4 160 Feb 4 12:34 RPMS
drwxrwxr-x 2 rjd4 rjd4 60 Feb 4 12:38 SOURCES
drwxrwxr-x 2 rjd4 rjd4 60 Feb 4 12:41 SPECS
drwxrwxr-x 2 rjd4 rjd4 40 Feb 4 12:33 SRPMS
```
Practical — compiling nano

- nano is a simple text editor a lot like pico
- Change directory to /tmp/userid/SPECS
- Create a spec file
  - nano.spec
  - Copy and adapt dummy.spec
- Complete it as far as the %build section
- Test it works
  - rpmbuild -bc nano.spec

Once you have the build tree ready you need to create a spec file. Instructions are provided for users of emacs and vi. You should not need to create any patches. The URL for nano is http://www.nano-editor.org/ and the source code can be found online at ftp://ftp.gnu.org/nano/nano-1.2.3.tar.gz. You should look at the package information for emacs and vim to determine what group to use.

$ cd /tmp/userid/SPECS
$ cp dummy.spec nano.spec

emacs:
$ emacs nano.spec &

vi:
$ vi nano.spec

*Test the spec file with rpmbuild -bp nano.spec,
*Test both the prep and build stages with rpmbuild -bc nano.spec.
Installation location

- “Build root”
  - Never install into the system tree
  - BuildRoot: _tmpath_/name/-version/-release-buildroot
- Default %install removes the build root
- There is a %{makeinstall} macro
- Rely on %{configure}
  - Uses DESTDIR variable in “make install”

Now we have compiled the code we must install it. However, we will not install it into the final “system” location. There are many reasons for this. Installing into the system directory tree (/usr/bin etc.) would interfere with any installed instances of the software and it also requires root privileges which we have so far avoided. Instead, rpm-build uses a “build root” — a copy of just that bit of the system directory tree needed for the package. The default %install scriptlet offered by the emacs template just gets rid of any existing tree to avoid contamination between installs. Clearly there needs to be support within the “make install” system to divert the installation from the system directory tree to the build root directory tree. There is a %{makeinstall} macro but it is not particularly useful. Makefiles built with configure have their install target divertible by setting the DESTDIR (“destination directory”) variable. This is prefixed on the front of all target directories for the installation phase, so that a file that would normally be installed as /usr/bin/xaos, say, is actually installed as ${DESTDIR}/usr/bin/xaos. We just need to set DESTDIR to the location of our build root.

%install
rm -rf ${RPM_BUILD_ROOT}
make install DESTDIR=${RPM_BUILD_ROOT}
Installation

- `rpmbuild -bi XaoS.spec`
- "-bi": Install
- Runs the %install section
- Removes any existing build root
- Installs into new one
- Manual shuffling
- Post-install activity

So now we can write the %install section. We remove any previously created build root to stop cross-contamination (something else we can't do if we use the system directory tree) and then install into a fresh one. configure-created Makefiles build any directories they need including the top-level build root itself.

```
%install
rm -rf ${RPM_BUILD_ROOT}
make install DESTDIR=${RPM_BUILD_ROOT}
```

Note that when we run "rpmbuild -bi" there is considerable activity after the two line scriptlet is complete.
Post-install activity

- /usr/lib/rpm/redhat/brp-compress
  - gzips certain files (man, info pages)
- /usr/lib/rpm/redhat/brp-strip
  - Strips unnecessary data from binaries
- /usr/lib/rpm/check-files
  - Checks for unpackaged files
  - %files section

As the final stage of the installation phase, rpmbuild performs some other actions:

- It compresses certain documentation files where it knows the corresponding documentation reading program can still use them in their compressed state. Any file that looks like a man page (in $DESTDIR/%{_mandir}) or an info page ($DESTDIR/%{_infodir}) will be compressed with gzip.

- On Red Hat systems, binary executables are “stripped”. This removes data from them that makes no difference to their execution but makes them a little smaller and less useful to debug with. The debugging information is moved elsewhere. (For example the povray executable changes from 1,578,360 bytes to 1,448,240.)

- Finally the files in the build root are compared with the list in the %files section. Obviously there is a large discrepancy at this point as our %files section is empty.
%files section

- List of every file to be included in package
- Must account for every file in build root
- Can “flag” special files
  - Documentation, configuration
- Default attributes
  - User, group, permissions
- Use macros to match %{configure}
- Quote directories for whole directory trees

The %files section needs to account for every file in the build root. Any file that is created in the build root by the %install script must be listed in %files. If any file is found in the build root that is not in %files then rpmbuild terminates with an error.

The listing serves another purpose: files can be flagged as being documentation, configuration files etc. We can also specify default and per-file ownerships and modes. This lets us define a setuid-root program in a package without ever having to be root to create the package. Of course, we have to be root to install the package for real.

When we list the files we should use the %{_location} macros to reflect where files have been put by the same macros within %{configure}. 
The XiaoS %files section

• 168 files!

%{bindir}/xaos
%{mandir}/man6/xaos.6.gz
%{infodir}/xaos.info.gz
%{datadir}/%{name}/catalogs
doc %{datadir}/%{name}/doc
doc %{datadir}/%{name}/examples
doc %{datadir}/%{name}/help
doc %{datadir}/%{name}/tutorial

If a directory tree is listed in the %files section it covers every file in that directory tree. This lets us use just six lines to account for all 3,356 files.

The %{bindir}/xaos line puts the /usr/bin/xaos binary in the package.

The %{mandir}/man6/xaos.6.gz line adds the compressed manual page. Recall that the compression is done by rpmbuild itself rather than our %build or %install scriptlets.

The %doc tag marks its argument as a documentation file. (More on this later.)

Given that we can quote directory trees in this list a novice might ask why we shouldn’t just give “/” as the sole line in the files listing. This would lead to a number of problems. First, it would have all the packages arguing over who owned /usr/bin, for example. Second, it would stop us labelling files as documentation or configuration.
%doc qualifier

- Specifies a file is documentation
- Can apply to directory trees of docs
- `rpm --install --excludedocs`

Why would we want to flag a file as a documentation file?

First, it is possible to install a package and to exclude the documentation. As disc becomes larger this is less relevant, but is still useful from time to time.

Second, it is possible to locate documentation files. If we use some of rpm's more advanced facilities, we can get a listing of all the files in a package together with the numerical value of their "flags":

```bash
$ rpm -q --queryformat '[%{filenames}\t{%{fileflags}}\n]' povray
/etc/povray.conf 1
/etc/povray.ini 1
/usr/bin/povray 0
/usr/share/doc/povray-3.50c 0
/usr/share/doc/povray-3.50c/README 2
/usr/share/doc/povray-3.50c/README.unix 2
```

0: No flags set
1: Configuration file %config
2: Documentation file %doc
9: File need not exist %config(missingok)
17: Never replace %config(noreplace)
64: File need not exist %ghost

We will address these flags in detail in the next session.

The flags are cumulative and the flag value of 9 for %config(missingok) should be interpreted as a value of 1 for %config plus 8 for (missingok).
%config qualifier

• Labels files as configuration files
• Expected to be modified by sysadmin
• Useful in --verify output
• “c” for configuration files

$ rpm --verify syslogd
S.5....T c /etc/syslog.conf

Again, why would we want to label a file as a configuration file?

First, we are more likely to modify a configuration file after installation than any other sort of file. Therefore when we come to verify a package it is reasonable to flag that a file is a configuration file at the same time as flagging that it has been changed.

Second, when we upgrade a package we don’t want to replace edited configuration files with the package’s original unless we have to. If a package upgrade was going to replace a configuration file with whatever the configuration file was before it was edited, it won’t. It is possible to be even more severe and instruct the system never to upgrade it.

A configuration file, thing.conf, has its default content, as provided by the package. Call this X. Over time the sysadmin edits it and it ends up as Y. If a package upgrade would provide the system with file X again, the file will not be touched if it is labelled as a configuration file. If the upgraded package would have installed file Z, though, it would normally move the file to a copy called thing.conf.rpmsave and set thing.conf to the new value Z. If the %config statement is replaced with %config (noreplace), though, then thing.conf will not be touched and Z will be written to thing.conf.rpmnew. Warning messages are written in both cases. A standard configuration file behaves like this:

• X–Z (i.e. no change to the package’s file): Version Y is left untouched.
• X!–Z, Y–Z (e.g. the administrator has brought in a fix that’s now the default): Version Y is left untouched.
• X!–Z, X–Y (i.e. The administrator made no change but the new package does): Version Z is installed.
• X, Y, Z all different: Version Y is renamed to have .rpmsave on its end and version Z is installed. The administrator is warned.
Building the package

- **rpmbuild -ba XiaoS.spec**
- 
- “-ba”: All
- Always works from scratch
- Creates two package files:
  - SRPMS/XaoS-3.1-1.src.rpm
  - RPMS/i586/XaoS-3.1-1.i586.rpm

And at last, we can make our package. The instruction is “rpmbuild -ba” (for “build all”). This will create the package from scratch (%prep, %build, %install) and then create the package. Note that to do this it does some final processing (just like %install is followed by the compression, stripping and file checking). We will examine what these checks are in a moment, but first we will look at the files we have created.

SuSE Linux defaults to creating binaries for the i586 (Pentium) architecture. Red Hat Linux defaults to the even older i386.
Source RPM

- SRPMS/XaoS-3.1-1.src.rpm
- *Everything* needed to rebuild the binary
  - rpmbuild --rebuild XaoS-3.1-1.src.rpm

The source package is a combination of the source files and spec file.

If we install this package with “rpm --install” then those files get put back into /usr/src/redhat, or whatever the top directory is at the time.

If we just want to rebuild the binary packages then “rpmbuild --rebuild” combines the installation of the source package and the “rpmbuild -ba” phase too.
Binary RPM

- The actual program and support files
- Everything you need to run the program
- Compiled for the generic “i586” architecture
- This is what the sysadmin installs

The package file in RPMS/i586 is the “real” package file. This contains all the binaries, configuration files, documentation, etc. This is all you need to run the program. The default compiler options have caused the program to be compiled for the oldest Intel platform supported by SuSE called “i586”.
Building an i686 package

- May want to exploit target-specific extensions
- Debatable benefits for general applications
- `rpmbuild --rebuild --target=i686
  XaoS-3.1-1.src.rpm`
- Creates
  `RPMS/i686/XaoS-3.1-1.i686.rpm`

We could build our package with the compiler knowing the exact target system. How much benefit we derive is an area of considerable controversy. We can build for the i686 (Pentium II) platform specifically by adding the option

`--target=i686` to either the `rpmbuild -ba` command or to the `rpmbuild --rebuild` command. Either way, a binary package file is created in `RPMS/i686` with the `.i686.rpm` suffix.

In many cases, it doesn't seem worth it based on the timings. For example, the ray-tracing program POV-Ray ships with a selection of test scenes. Building the advanced set with the i386 binary took 51 minutes. With the i686 binary it took 48 minutes, well within the error bounds.
Requirement analysis

- Last stage of package build
- Requires: ...
- Checks packaged files for requirements
- Shell scripts need shells
- Binary files need libraries

Now let's consider some of the other lines that might come out of the build process.

Requires(interp): /bin/sh

Somewhere in the collection of files is a shell script whose interpreter is /bin/sh. So for this file to work, the package requires /bin/sh to be provided by the system.

Requires: ld-linux.so.2 libaa.so.1 libc.so.6 libc.so.6(GLIBC_2.0) libc.so.6(GLIBC_2.1) libc.so.6(GLIBC_2.1.3) libc.so.6(GLIBC_2.3) libdl.so.2 libgpm.so.1 libm.so.6 libm.so.6(GLIBC_2.0) libcurses.so.5 libpng.so.3 libslang.so.1 libX11.so.6 libXext.so.6 libz.so.1

The executable binaries in the package use these shared libraries for some of their functionality. We will discuss this in detail next.
Runtime library requirements

- Automatically determined by rpmbuild
- Library features can be labeled
  - Features introduced at certain versions:
    - lib.so.6
    - lib.so.6(GLIBC_2.0)
    - lib.so.6(GLIBC_2.1)
    - lib.so.6(GLIBC_2.1.3)
    - lib.so.6(GLIBC_2.3)
- C library must be version 2.3 or later

rpmbuild launches a program that searches through the packaged files finding all the binary executables. These are checked to see what libraries they use. This process is slightly more sophisticated than the ldld program; a well-written library attaches tags to the functions it provides identifying what version of the library the function was introduced in. So the library requirement finder not only spots that we use the GNU C library (lib.so.6) but that we use some functions that were introduced in version 2.0 and others introduced in version 2.1, version 2.1.3 and version 2.3.
Practical — packaging

• Complete the %install section
  - rpmbuild -bi nano.spec
• Take a look in the build root
• Complete the %files section
  - Don't forget %config and %doc
  - Use macros
• Build your package for the i686 architecture
  - rpmbuild -ba nano.spec

And now it's your turn. You have now seen everything required to create a package. You should now complete your nano package.
Second practical — installation

- root password
  - On the whiteboard
  - For this course only!

```bash
rpm --install
/tmp/userid/RPMS/i686/nano-1.2.3-1.i686.rpm
```

- Run nano

Don’t expect this password to ever work again.

And please don’t do anything silly with the root password. It’s rude.
Improving the package

• Next release!
  - Increment release number for a new build
• Improvements?
  - Improved %description section
  - Verification script
  - Explicit dependencies?

Now we've made a package, let's make a good package. We're now going to make a second release (so increment the release number in the spec file) which will have a set of requirements which is better than the automatically derived ones provided automatically. We will also bolster our %description section and consider what a good % description should contain. Finally we will add a facility to make it easier to test packages.
A better %description

- Home URL already listed
- Plain text
- Text from web site?
- Reference to local documentation
  - local web pages
  - man page
  - info page

How do we make the description better? There should be some minimum standard for this section, but there isn’t, and some distributions make barely any effort at all. Before we start it should be noted that the section must be plain text; we can’t use HTML unfortunately.

Clearly there should be a description of the package and this should be more than the one line summary. My personal rule of thumb is that the summary should be the hint that say “this is the package you want” to the administrator who knows the product and that the description should describe the software to the administrator who doesn’t. I find that taking the text from the software’s website’s front page often does the trick.

The other item the description should contain is a reference to the local documentation. Is the curious user better off looking for a man page, an info page or is there a full HTML documentation directory on the local system? Explicit reference should be made.
XaoS %description

- Text from web page
XaoS %{version} is a fast real-time interactive fractal zoomer. It displays the Mandelbrot set (among other escape time fractals) and allows you to zoom smoothly into the fractal. ...

- Local documentation
Try "man xaos" or "info xaos" for instructions or use the built-in tutorial system.

- Macros can be used in this section

For the XaoS package, I take the text from their web site http://xaos.sourceforge.net/ and add a paragraph directing the curious user at the manual page and the built-in tutorial.

Note that macros can be used here too.
Verification script

- Applies to the installed package
- `rpm --verify`
  - Tests file properties
  - Can run a script too
- `%verifyscript`

The `rpm` program has a `--verify` option which checks various properties of a package's files. However, it can be extended to run a test script too. This script is introduce with the `%verifyscript` directive.

Maintainers of "serious" packages might like to consider this as a vehicle for running regression tests to track fixes to bugs. Alternatively they might like to create a "--test" subpackage (we will cover subpackages in the next session) to do this.
XaoS: `%verifyscript`

- XaoS can be run non-interactively
- Can't run the GUI from `%verifyscript`
  - It may not be supervised
- Thoroughness vs. time taken
- Script may get run as `root`
  - Take extreme care

XaoS can be run without the GUI appearing, allegedly. We can't use a GUI from the verification script because there might not be a display and it might be run unattended. The whole verification process might get held up waiting for someone to close the application window.

One word of warning: the verification process may be run by the system administrator as `root`. Be careful how you write the script.
Explicit dependencies

- Automatic requirement determination
- Dependencies on
  - Libraries
  - Interpreters
- What package provides these?
- NB: Very distribution dependent

At the moment, when we generate our package `rpmbuild` generates an automatic set of requirements, mostly libraries and interpreters (most typically `/bin/sh`).

`Requires: ld-linux.so.2 libaa.so.1 libc.so.6 libc.so.6(GLIBC_2.0) libc.so.6(GLIBC_2.1) libc.so.6(GLIBC_2.1.3) libc.so.6(GLIBC_2.3) libdl.so.2 libgpm.so.1 libm.so.6 libm.so.6(GLIBC_2.0) libncurses.so.5 libpng.so.3 libslang.so.1 libX11.so.6 libXext.so.6 libz.so.1`

Now consider this from the point of view of the installing system administrator. “It’s all very well saying I need the `libncurses.so.5` library, but what package has it?”

For the time being, I recommend putting the onus on the installing system administrator to know his system packages. The situation might be different if we were depending on non-system packages. We will address these later.
Don't forget the change log!

- Keep track of changes
- Well worth it
  - Makes support easier
  - Looks professional

%changelog
* Wed Jan 28 2004  <rjd4@cam.ac.uk>
- Improved description section
- Verification script

* Tue Jan 27 2004  <rjd4@cam.ac.uk>
- Initial build.

Finally, there's the change log. This is often regarded as a frill, but it genuinely does make it easier for people to track what's going on and is a good discipline to get into.
Release two!

- Better user support
  - %description
- Verification script
  - %verifyscript

So now we are ready for release two!
Practical

• Make improvements to your nano package
• Build release 2
• Upgrade the installed package
  – As root
    rpm --upgrade /tmp/userid/RPMS/i686/nano-1.2.3-2.i686.rpm
• Check the new package information
  – As the user
    rpm --query --info nano

The third practical, and the final one of this session, is for you to make similar changes to your nano package. When you are done, save your nano.spec file in your home directory. /tmp gets cleared between reboots.
Second day

• Binary-only software
• Multiple packages from one source
• Adding to the application menu

In this second half we are going to look at the hard stuff now that we have met the principles.

Not all software is released as source code. We want to be able to package this software too. That won’t turn out to be hard, but varies depending on the software in question. We will use Sun’s Java distribution for this.

The Java distribution comes in various bits so we will then investigate how to create multiple binary (installable) packages from a single source package. This will let us pick and choose which bits we get.

We will also work out how to get items into the applications menus corresponding to these applications.
Binary-only software

- Gratis software, but no source code
  - e.g. java, condor, ...
- Often not packaged the way we want
- Needs converting into a package
- Sun's java distribution
  - Ships as "self extracting binary"
  - j2sdk-1.4.2_05-linux-i586.bin
  - Prompts for accepting licence

Sun's Java distribution for Linux comes in two forms: a self extracting binary and a binary RPM. The RPM, however, is not very good and we can do better. So we will start with the simpler distribution and wrap it in our own packaging infrastructure. First we must see what the self-extracting binary actually does.

```sh
$ ./j2sdk-1.4.2_05-linux-i586.bin
Sun Microsystems, Inc.
Binary Code License Agreement
for the
JAVATM 2 SOFTWARE DEVELOPMENT KIT (J2SDK), STANDARD EDITION, VERSION 1.4.2_X

For inquiries please contact: Sun Microsystems, Inc., 4150 Network Circle,
Santa Clara, California 95054, U.S.A.
(LFI#135003/Form ID#011801)

Do you agree to the above license terms? [yes or no] yes
Unpacking...
Check summing...
0
0
Extracting...
UnZipSFX 5.40 of 28 November 1998, by Info-ZIP (Zip-Bugs@lists.wku.edu).
  creating: j2sdk1.4.2_05/
    creating: j2sdk1.4.2_05/jre/
```

And the output is paged through the more program. So to get it to unpack we must disable its use of more and feed it a “yes” answer:

```sh
$ echo yes | ./j2sdk-1.4.2_05-linux-i586.bin > /dev/null
```

If we can't script the unpacking then there's no point proceeding.
Getting started

- Package name: j2sdk
  - “Java 2 Software Development Kit”
- Fill in the template spec file
- j2sdk has an executable unpacker
- %prep should make sure it's executable
  - %{S:0} expands to the “Source0:” file
- %build is null

So let's proceed. We'll create a package called j2sdk with the version number provided by Sun (1.4.2_03) and release number 1.

Source files are not normally executed directly, so we will use the %prep section to make sure it is executable. Note the macro trick that lets us refer to the various source files. %{S:N} expands to the Nh source file, i.e. The one declared with the “SourceN:” line. There is a corresponding %{P:N} to refer to the Nh patch file.

%build has nothing to do.

Name: j2sdk
Version: 1.4.2_05
Release: 1
License: Sun
URL: http://java.sun.com/
Source0: %{name}-%{version}-linux-1586.bin
BuildRoot: %_tmppath%/%{name}-%{version}-%{release}-buildroot
Summary: Java 2 Software Development Kit
Group: Development/Languages

%description
The software development kit and runtime system for Sun's Java 2 language.

%prep
chmod a+rx %{S:0}

%build
We will run the self-extractor in the %install section.
Installing j2sdk

• Install into build root
• Investigate what actually unpacks
  - j2sdk1.4.2.05 in current working directory
• Must make directories manually
  - "mkdir --parents" builds intermediate directories
• Run installer in build root
  - "> /dev/null" disables paging
  - "echo yes |" answers the licence question

The self-extracting binary creates a directory j2sdk1.4.2.05 in the current working directory. We will decide that we want Java installed as %(_prefix)/java/j2sdk1.4.2.05 so we must make the necessary directory structure (including the build root itself) manually. This is where the "--parents" option on mkdir helps as it builds the intermediate structure. We change directory to the %(_prefix) subdirectory of the build root and run the script from there.

%install
rm -rf ${RPMBUILD_ROOT}
mkdir --parents ${RPMBUILD_ROOT}$_prefix]/java
cd ${RPMBUILD_ROOT}$_prefix]/java
echo yes | %{S:0} > /dev/null

And that completes the installation of everything provided by Sun's self-extracting binary.
Listing the files

- Could get away with one line!
  - \%{_prefix}/java/%{name}\{_version\}

- That's a cop out
  - Doesn't identify documentation etc.

As everything we have unpacked belongs in a single directory tree we could get away with a single line %files
section identifying that directory. But this is a cop out; we have established various attribute modifiers that we
might want to apply, so let's apply them.

%files
%defattr(-,root,root,-)
%dir {%_prefix%}/java/%{name}\{_version\}
%doc {%_prefix%}/java/%{name}\{_version\}/COPYRIGHT
%doc {%_prefix%}/java/%{name}\{_version\}/LICENSE
%doc {%_prefix%}/java/%{name}\{_version\}/README
%doc {%_prefix%}/java/%{name}\{_version\}/README.html
%doc {%_prefix%}/java/%{name}\{_version\}/THIRDPARTYLICENSE/README.txt
%doc {%_prefix%}/java/%{name}\{_version\}/man
%_prefix%/java/%{name}\{_version\}/bin
%_prefix%/java/%{name}\{_version\}/demo
%_prefix%/java/%{name}\{_version\}/include
%_prefix%/java/%{name}\{_version\}/jre
%_prefix%/java/%{name}\{_version\}/lib
%_prefix%/java/%{name}\{_version\}/src.zip
%_prefix%/java/%{name}\{_version\}/.systemPrefs

This gives a greatly higher quality of package.
Library requirements

- Some “required” libraries are in the package itself
  - rpmbuild only finds some of them
  - Provides: c.f. Requires:
- Some “required” libraries aren’t required
  - This can cause problems
  - Some software degrades gracefully if support libraries are missing
  - rpmbuild doesn’t know this

At the end of the installation phase the usual requirements checks are done and these throw up some oddities.

Requires: /bin/bash /bin/sh libX11.so.6 libXext.so.6 libXi.so.6 libXp.so.6 libXt.so.6 libXtst.so.6 libasound.so.2 libasound.so.2(ALSA_0.9) libawt.so libc.so.6 libc.so.6(GLIBC_2.0) libc.so.6(GLIBC_2.1) libdl.so.2 libdl.so.2(GLIBC_2.0) libdl.so.2(GLIBC_2.1) libgcc_s.so.1 libgcc_s.so.1(GCC_3.0) libjava.so libjava.so(SUNWprivate_1.1) libjvm.so libjvm.so(SUNWprivate_1.1) libm.so.6 libm.so.6(GLIBC_2.0) libm.so.6(GLIBC_2.1) libmlib_image.so libnet.so libnet.so(SUNWprivate_1.1) libnsl.so.1 libodbc.so libodbcinst.so libpthread.so.0 libpthread.so.0(GLIBC_2.0) libpthread.so.0(GLIBC_2.1) libverify.so libverify.so(SUNWprivate_1.1)

Some of the required libraries are provided by the package itself. For some reason rpmbuild hasn’t found them. For these we will add a line to the spec file explicitly stating that they are provided (analogous to the Requires: lines). In our case these are easily spotted by the fact that they come with the label “SUNWprivate_1.1”.

On some Linux distributions some of the required libraries are system libraries which we don’t have. This is a common side-effect of binary installs. One of three things can happen here:
(i) the software really needs this library and we must ensure at install time that the relevant library is on the system,
(ii) the software really needs this library and it’s not available, in which case we drop the whole packaging project because the software itself can’t run, and
(iii) the software doesn’t really need these libraries and degrades gracefully in their absence.
The “not really missing” libraries

- rpm thinks we are missing:
  - libjava.so
  - libnet.so
  - libverify.so
- Labelled as SUNWprivate_1.1
- Add explicit Provides: lines
  - Provides: libjava.so(SUNWprivate_1.1)
  - Provides: libnet.so(SUNWprivate_1.1)
  - Provides: libverify.so(SUNWprivate_1.1)

We should check that the libraries really are provided before we add the Provides: lines.

```
$ cd /var/tmp/j2sdk-1.4.2_05-1-buildroot/
$ find . -name libjava.so -print
./usr/j2sdk1.4.2_05/jre/lib/1386/libjava.so
$ find . -name libnet.so -print
./usr/j2sdk1.4.2_05/jre/lib/1386/libnet.so
$ find . -name libverify.so -print
./usr/j2sdk1.4.2_05/jre/lib/1386/libverify.so
```

They are provided and in our Provides: lines we should add the labels that they support.
Release 1!

- Now we can build the package
- Release 1!
  
rpmbuild -ba j2sdk.spec

$ rpmbuild -ba j2sdk.spec
Wrote: /usr/src/redhat/SRPMS/j2sdk-1.4.2.05-1.src.rpm
Wrote: /usr/src/redhat/RPMS/i586/j2sdk-1.4.2.05-1.i586.rpm
_
Splitting up the package — 1

• What if we don't want
  – the demo software?
  – the library source code?

For release 2 we will start to split up the package.

The demo software directory and the library source code each occupy slightly over 10% of the entire space occupied by the Java installation. But they might be regarded as frills. They occupy disc, though disc is cheap, and they take time to install, and bandwidth isn't. It is quite plausible, then, to envisage a situation where we don't want those two elements installed.
Splitting up the package — 2

- Build three packages at once:
  - j2sdk
  - j2sdk-src
  - j2sdk-demo
- Installer gets to pick and choose

What we will do is to build three packages at once. The installing system administrator can then pick and choose between the elements.
Release 2 of the j2sdk

- Split into three packages
  - Main package
  - Two “sub-packages”
  - “name-suffix” convention
- Select different groups for the packages
- Different summaries for the packages
- Different %descriptions for the packages
- Different %files for the packages

Some of the parameters we set for a package will vary between the four packages. The groups may vary, the summaries and descriptions ought to vary and the files lists will have to vary. A file cannot be in more than one package.
Specifying the groups/summaries

- First (usual) set of headers:
  - specify the main package
- `%package suffix`
  - introduce the sub-packages
  - Group:
  - Summary:

We will start at the top of the spec file with the standard headers. We will split out two, Group: and Summary:, and set them differently for each package. The pair of lines as they first appear correspond to the “base package”, j2sdk. After that a `%package` directive can specify that the following lines correspond to a sub-package. Note that `%package` takes the sub-package’s suffix and not its whole name. This reinforces the `base-suffix` naming structure.

```bash
# Base package
Group: Development/Languages
Summary: Java 2 software development kit
Provides: libjava.so(SUNWprivate_1.1)
Provides: libnet.so(SUNWprivate_1.1)
Provides: libverify.so(SUNWprivate_1.1)

%package demo
Group: Applications/Examples
Summary: Java 2 demo software

%package src
Group: Development/Languages
Summary: Java 2 library source code
```

I have put the Provides: header in its new position to emphasise that it only applies to the base package and the sub-packages could provide (and require) different things.

If I was doing this for real I would probably unpack the src.zip file and package up its unwrapped form directory tree.

For completeness I should note that it is possible to specify a whole package name with a spec file instruction along the lines of

```bash
%package -n full_package_name
```

but it’s better to stick with the suffix model.
Specifying the %descriptions

- %description
  - Specifies the main package
- %description suffix
  - Specifies the subpackages

Next we define the descriptions for the packages.

This is done with a variant form of the %description command rather than by having the sub-package's particular %description follow its %package directive:

%description
The software development kit and runtime environment for Sun's Java programming language.

%description demo
Demonstration software to illustrate the features of the Sun Java SDK version %{version}

%description src
A zipped archive of the Java source code to the base class libraries.
Specifying the %files

- %files
  - Specifies the main package
- %files *suffix*
  - Specifies the subpackages

And finally we have the files:

```
%files
defattr(-,root,root,-)
dir %{prefix}/java/%{name} %{version}
doc %{prefix}/java/%{name} %{version}/COPYRIGHT
doc %{prefix}/java/%{name} %{version}/LICENSE
doc %{prefix}/java/%{name} %{version}/README
%doc %{prefix}/java/%{name} %{version}/README.html
doc %{prefix}/java/%{name} %{version}/THIRDPARTYLICENSE
%doc %{prefix}/java/%{name} %{version}/man
%{prefix}/java/%{name} %{version}/bin
%{prefix}/java/%{name} %{version}/include
%{prefix}/java/%{name} %{version}/jre
%{prefix}/java/%{name} %{version}/lib
%{prefix}/java/%{name} %{version}/.systemPrefs

defattr(-,root,root,-)
%{prefix}/%{name} %{version}/demo
defattr(-,root,root,-)
%{prefix}/%{name} %{version}/src.zip

%files src

defattr(-,root,root,-)
%{prefix}/%{name} %{version}/src.zip

Note the repeating default attributes line.
Inter-package requirements

- Does j2sdk-src require j2sdk?
  - Not really.
- Does j2sdk-demo require j2sdk?
  - Yes.
  - That's what it's demoing.
- Add Requires: lines under %package

  %package demo
  Requires: %{name} = %{version}

We're almost done.

There are explicit dependencies between the sub-packages and the base package. We can add explicit requirements for some of our sub-packages by adding Requires: lines to the %package sections. Note that we can cause the dependency to be on a specific version of the package.
Building the split packages

• Same build instruction as usual:
  - rpmbuild -ba j2sdk.spec

• Builds more packages:
  - SRPMS/j2sdk-1.4.2.05-2.src.rpm
  - RPMS/i386/j2sdk-1.4.2.05-2.i386.rpm
  - RPMS/i386/j2sdk-demo-1.4.2.05-2.i386.rpm
  - RPMS/i386/j2sdk-src-1.4.2.05-2.i386.rpm

And now we are done. We have release 2.
Practical

• Split the nano package
  - nano
  - nano-docs
• nano to contain manual and info pages
• nano-docs to contain the other docs

The practical is to make a split nano package which has the run-time system (and manual page) in the primary package and all the other documentation (and FAQ and some other bits) in the other.
Release 3

• java still not on the PATH
  – Problem with release 2
  – Change the PATH?
  – Add links in /usr/bin?

• Good thing about release 2:
  – Possible to install multiple versions
  – We don't want to lose this

We still haven't quite got it right. At the moment the java executable is tucked away in the file
/usr/java/j2sdk1.4.2_03/bin/java. That's not somewhere where the user can find it. We can do one
of two things: either we can adjust the user's PATH so that the shell looks for programs in the directory
/usr/java/j2sdk1.4.2_03/bin or we create symbolic links or wrapper scripts from a directory in the
user's PATH to the other locations.

At the moment we can install multiple versions side by side because everything comes in a directory with the
version number wired into its name. Given how fast a moving target Java is, we might want to maintain this
flexibility.
Adding an entry to the PATH

- /etc/profile.d
  - A directory of files sourced by login shells
  - But *only* login shells
  - Can also set other environment variables
- What about multiple versions installed?
  - Make PATH-changing a distinct sub-package
  - j2sdk-path?

To change the PATH for every user, add files to /etc/profile.d. Each time a user logs in using a Bourne-style shell, all the *.sh files in that directory are sourced. Furthermore, we can also set other environment variables this way if it's useful.

If we want to have multiple versions installed then the package that puts the file(s) in /etc/profile.d should be a separate package, j2sdk-path say, so that multiple versions can be installed and one can be selected as the default.
Adding entries to /usr/bin

- Already on the PATH
- Symbolic links to “real” location
- Multiple versions?
  - Make a distinct sub-package
  - j2sdk-links?
- /usr/bin/jar
  - clashes with libgcj
  - much harder problem to fix

The alternative is to add entries to /usr/bin (or some other directory on the PATH). Again, this would require a separate sub-package to allow us multiple versions. A more serious problem, however, is that /usr/bin/jar is already spoken for and there may be clashes with other packages.
Environment setting

- We will implement the PATH approach
- Need to add
  - /etc/profile.d/j2sdk.sh
  - /etc/profile.d/j2sdk.csh
- Extra source files?
  - Simple enough to build in %build
  - Wire in the version number at build time

So we'll implement the PATH-modifying approach in release 3.

We need to add two files, one for Bourne shell users and another for the poor souls who choose to use C-shell. As we want to wire in the version number (which forms part of the path to the binaries' directory) we won't use fixed files but will create them at build time so that we don't have to change them every time we build the package for a new version of Java. Alternatively we could have had a pair of source files and run sed on them in the build phase to insert the version number.

%package path
Group: Development/Languages
Summary: Put version %{version} of the Java 2 SDK on the PATH
Requires: %{name} = %{version}

%description path
The files needed to make version %{version} of the Java 2 software development kit appear on the users' default PATHs.

%build
printf 'export PATH=%s:${PATH}\n' \%prefix\}/java/%{name}\%{version}/bin > j2sdk.sh
printf 'path=(%{ path}\n' \%prefix\}/java/%{name}\%{version}/bin > j2sdk.csh
chmod u=rw,g=rx j2sdk.sh j2sdk.csh

%install

- mkdir --parents $(RPM_BUILD_ROOT)%{_sysconfdir}/profile.d
cp j2sdk.sh j2sdk.csh $(RPM_BUILD_ROOT)%{_sysconfdir}/profile.d/.

- %files path
  %{_sysconfdir}/profile.d/j2sdk.*
Release 3

- We now build another release!
  - Release 3.
  - Yet another sub-package.
  - Seeing a pattern yet?
- What's next?

You may have noticed that as we add to the Java 2 packages the number of sub-packages increases. This is not an accident. Packages are the units of installed software; as the available detail increases so does the number of packages.
Adding items to the menu

- Applications launched from the menu
- Menus vary between distributions
- Still possible to define a menu configuration
- We will add an item for a Java demo program

The next stage of building the package is to integrate the application into the windowing system's menu system. We are told (though I don't believe it) that users can't cope with the command line any more and need to click on icons to make things happen.

The menu structures can vary quite dramatically between distributions, but underlying this there is a common mechanism to set things out.

In our case we will add a menu item to launch one of the Java demo programs. It will belong in the j2sdk-demo package, therefore.
Structure of the “start” menu

- Variation between distributions:
  - Red Hat 9: Accessories ➔ Calculator
  - SuSE 9: Applications ➔ Utilities ➔ Calculators ➔ Calculator
- Two different desktop systems:
  - GNOME
  - KDE
- Can still use a common configuration file
- Situation confused by distribution “specials”

Menus are one of the things that varies most between distributions. For example, Red Hat’s “start” menu has various categories of application listed directly, with various special operation beneath them. SuSE, on the other hand, has a very short menu with an applications sub-menu corresponding to the top part of Red Hat’s. What Red Hat calls “accessories”, SuSE calls “utilities”. And so on... To make matters even more interesting there are two different commonly used desktop environments called GNOME and KDE (http://www.gnome.org/ and http://www.kde.org/ respectively).

Nonetheless we can still integrate applications into the menus in a cross-distribution, cross-desktop way. We do this not by specifying exactly where the item should go in the menu but by describing the type of application and letting the windowing system decide. This is managed by the files in /usr/share/applications with the suffix .desktop. This work is being standardised by the Free Desktop organisation (web pages at http://www.freedesktop.org/).

I’m painting an overly rosy picture here. The situation is complicated by the distributions’ control freak tendencies but we will start with the default behaviour and demonstrate variations on the theme later.
Application categories

- Menu structure dynamically generated
  - Rules vary between distributions
- Based on application’s “desktop file”
- Desktop file sets application’s “categories”
- System uses categories to place menu item

The key to placing an application in a menu is to identify in its desktop file what “categories” it belongs in. There are 115 categories currently defined (http://www.freedesktop.org/standards/menu-spec/0.8/apa.html) which interconnect, so an application in the Email category is almost certainly in the Network and probably Office categories too. All three categories should be quoted.

We can look at some existing desktop files to see what’s going on. For example, here is the desktop file for the gimp graphics manipulator from a SuSE 9.0 system:

[Desktop Entry]
Encoding=UTF-8
Name=Gimp
Exec=gimp %F
Icon=gimp
Type=Application
GenericName=Image Manipulation Program
MimeType=image/gif;image/jpeg;image/bmp;image/tiff;image/png;
X-KDE-StartupNotify=false
X-StandardInstall=true
Categories=Graphics;RasterGraphics

Consider the Categories= row. There are two entries here, but “Graphics” is the one we are most interested in.
Desktop file

- Header
- Encoding
- Name
- Comment
- Exec
- Icon
- Categories
- Type
- Terminal

These are the rows we are interested in and really must specify:

The file must start with the line “[Desktop Entry].”

This must be followed by the encoding line which specifies the character set used in the rest of the file. The world seems to be converging on UTF-8. Don’t buck the trend.

Encoding=UTF-8

Next comes the name of the application. This is the text that appears in the menu item. In addition to specifying Name=Java 2D demonstration
we can specify the same in other languages for people who have a locale set:
Name[es]=Java 2D demostración

We can also specify a “comment”. This typically appears in a tooltip popup if the pointer floats over the menu item long enough (and if tooltips have not been disabled by the user). Again we can give language-specific comments:
Comment=A demonstration of Java 2D
Comment[es]=Una demostración de Java 2D

The Exec=... line specifies the command to be run. We will give the long command line
Exec=/usr/j2sdk1.4.2_05/bin/java -jar
    /usr/j2sdk1.4.2_05/demo/jfc/Java2D/Java2Demo.jar

The Icon=... line specifies an icon to appear alongside the text. We will use the only icon shipped with the Java distribution
Icon=/usr/j2sdk1.4.2_05/jre/plugin/desktop/sun_java.png

We will pick the “Utility” and “Amusement” categories for our application. Note the trailing semi-colon.
Categories=Utility;Amusement;

The next line, Type=, is where we should specify that this is an application. The desktop file is used for more than just the menu so this is not quite as redundant as it sounds. Valid types are application, links, file system devices and directories. But we want
Type=Application

Finally we specify whether the application should be launched inside a terminal (xterm etc.) It shouldn’t.
Terminal=false
Java 2D desktop file

- Create a desktop file
  - java2d.desktop
  - j2sdk-demo package
- Second source file!
  - Source0: %{name}-%{version}-linux-i586.bin
  - Source1: %{name}-java2d.desktop
- Use sed to substitute in the version number

So let’s create the desktop file. We know what we want in the file, but there are two issues we must address. The first is that this is a second source file where until now we’ve only had one. The second is that we might want to avoid having to change this source file every time we change Java version (and hence pathname).

Adding a second source file to the build is simple. Simply add another SourceN: line. We can refer to the Nth source file with the macro construction %{S:N}, so the binary can be referred to as %{S:0} and the desktop file as %{S:1}. Note that I’ve set the source file name to begin with “j2sdk”. This is just for maintaining sanity in the SOURCES directory and is not required by rpmbuild.

This source file will not be exactly what gets installed, though. We will replace the Java version “1.4.2_03” with the string JAVA_VERSION in the source file and used sed to insert the version for this package as it is built. This means that we don’t need to change the source file when we build the next package. (Unless Sun rename or withdraw the demonstration)

Source1: %{name}-java2d.desktop

%build
  _ sed -e s/JAVAVERSION/%{version}/ < %{S:1} > java2d.desktop
%install
  _ mkdir --parents %{rpm_build_root}%{_datadir}/applications
  _ cp java2d.desktop %{rpm_build_root}%{_datadir}/applications/.

The source file j2sdk-java2d.desktop will look like this:

[Desktop Entry]
Encoding=UTF-8
Name=Java 2D
Comment=Java 2D demonstration
Exec=/usr/java/j2sdkJAVA_VERSION/bin/java -jar
   /usr/java/j2sdkJAVA_VERSION/demo/jfc/Java2D/Java2Demo.jar
Icon=/usr/java/j2sdkJAVA_VERSION/jre/plugin/desktop/sun_java.png
Categories=Utility;Amusement;
Type=Application;
Terminal=false
Practical

• Arrange for nano to be on the menu
• Categories:
  - Utility
  - TextEditor
• Icon:
  - nano-icon.png

And now you do nano. I have provided an icon for you as a second source file which will have to be installed somewhere. (/usr/share/icons/pico.png, /usr/share/pico/icon.png, or somewhere else all together. It’s your choice. You will have to write the desktop file from scratch for your third source file.
“Day three”

- Follows directly on from “Building RPMs”
- You must have attended that course.
- Continuing with the Java package j2sdk

This is the “third afternoon” of a two afternoon course that grew beyond its bounds. As a result it follows directly from day two of the building RPMs course and will continue with the spec file of the binary-only j2sdk package. If you weren’t in on the previous course you will not understand much of what you are about to see.
Installing a system service

- j2sdk contains orbd
  - **Object Request Broker Daemon**
- Designed (poorly) to be run permanently
- Needs integration into the system services
  - Launch at boot
  - User to run as
  - Logging
- Another sub-package!

Part of the software provided by the Java 2 package is the Object Request Broker daemon, orbd. This is (poorly) designed to be a system service providing a function over the network. Don’t worry about the jargon; it’s a network server daemon. As such it needs to be launched at boot time, which requires a particular sort of startup script which the system will need to be configured to run, it needs a user to run as who is not root (who runs the boot sequence) and it needs to log somewhere. Guess what? We’re going to have another sub-package!
The orbd program

- Runs non-stop
- Does not background itself
- It needs some port numbers specified
  - It has built-in defaults.
- It needs a database directory
  - Uses ./orbd.db as a default
- Logs to standard output

Before we launch it we need to know what it needs. It runs non-stop without backgrounding itself (so arguably should not be called a “daemon”) and listens on a number of network ports which can be specified on its command line (if you want to override the defaults). It maintains a database of state in a directory that can also be specified on the command line and which defaults to a subdirectory of its current working directory. Finally, it generates its logging output on standard output.

We will need to create a configuration file that can specify the orbd options, the database directory and the log file.
The j2sdk-orbd package

- Leave the orbd binary where it is
  - /usr/j2sdk1.4.2_05/jre/bin/orbd
  - Base j2sdk package
- Package to *launch* the service
  - Configuration file
  - Start up script
  - Logging
  - Dedicated user

We're not going to pull the binary that is the daemon out of the base package. Firstly, we don't need to and the package should reflect the base software distribution as closely as possible. Secondly, it will just make our %files sections even longer and we don't need the hassle. All this means is that the sub-package will be responsible for the *launch* of the daemon rather than its *presence* and will depend on the base package to provide it. This will lead to a Requires: line like we've seen before:

Requires: %name = %version

So “all” our package will do is set up launching, make sure there is logging (which requires log rotation), create a user for the daemon to run as, and make sure there is a comprehensive configuration file for the administrator who wants to override our defaults.

All this is fairly typical, though the logging from the daemon is atypically poor.
New sub-package

• %package orbd
• Group: System Environment/Daemons
• Summary: Java 2 Object Request Broker Daemon
• Requires: %{name} = %{version}
• %description orbd
  This package launches the Java 2 object request broker. An ORB ...

The description of the sub-package is straightforward.
orbd scripts

- This is a course about Linux packaging
  - Not a course about Linux boot sequences
  - Not a course about Linux log rotation
- Take the scripts as given
  - Start up
  - Log rotation
- Still plenty of packaging details to cover

In your SOURCES directory are a collection of scripts for the sub-package. It is important not to get too distracted by their content. This is a course about packaging, not about the Linux boot sequence nor about Linux log rotation. Discussion of these scripts will be restricted to the elements required for packaging. Do feel free to look them over in your own time if you want to understand more about the various processes involved.
Start up/shut down script

- Script contents vary between distributions!
  - Two sub-packages?
  - Set parameters in a configuration file
- Script installed in directory
  %(_initrddir)
- Configuration files installed in directory
  %(_sysconfdir)/sysconfig
- Put the version/path in the script
  - configuration file constant between versions

Linux uses a common script for starting and stopping a service. The styles of the scripts vary between distributions. The SuSE distribution has a rather more sophisticated support structure for its scripts than Red Hat, say, which we would like to use where it is available. This means that we need to have both styles of script in our source RPM and select between them when we build our binary RPMs. We could use a single script — the SuSE and Red Hat styles are close enough — but it's always a good move to make your scripts as close to what the system expects as possible.

We will ship startup files for both Red Hat and SuSE and in a perfect world we would build two sub-packages, one to launch the daemon for a SuSE system and the other for Red Hat systems. However, because a file can only be assigned to a single (sub-)package during the build process we can't do that simply.

We will also create a configuration file that will set various parameters used in the startup script. We need to select what goes in the configuration file and what goes in the script so that the configuration file changes as infrequently as possible between versions of the package. So in our case the configuration file should not contain anything that contains the binary path name because that contains the version. Anything that the system administrator might want to change, though, definitely should stay out of the startup script and go in the configuration file.

There are a few tricks that are useful in the context of separate (non-configuration) scripts and configuration files.

NB: There is a bug in SuSE 9.0 whereby the %_initrddir macro is incorrectly defined to be /etc/rc.d/init.d rather than /etc/init.d. The bug is fixed in SuSE 9.1. You can work around this bug by correctly defining the macro in your ~/.rpmmacros file:

%_topdir /tmp/rj4d
%_initrddir /etc/init.d
Extra source files

- SOURCES/j2sdk-orbd-suse.sh
- Detail of contents irrelevant
- Third source file:
  - Source2: %{name}-orbd-suse.sh
  - Source3: %{name}-orbd-redhat.sh
  - %{S:2}, %{S:3}
- We won't use %{S:3} here
  - Same again for a Red Hat launch sub-package

So, we need to pack some extra source files. We do that simply by adding extra SourceN: lines. Even though only one of them will be used in any specific binary distribution, both will be included in the source package. We get at them with the %{S:N} macro.
Useful tricks

- Wire paths into the script with sed
- Write the configuration file from scratch
- Exploit shell substitution syntax
- Explicitly test the necessary parameters
- Use the distribution’s launch program
  - Red Hat: daemon shell function
  - SuSE: startproc program

First we have to get paths into the script and the configuration file which arise from the rpm macros. The scripts have large amounts of text in them of which we want to modify only a little. So we will write the source scripts with strings like JAVA_VERSION in them and during the %build phase use sed to replace those with the values given by the assorted macros.

Within the startup file itself there is a trick that can be applied to the options of the launched daemon. If we have a variable FOO set whose value we want passed to the daemon’s --foo option then we obviously want something along the lines of ‘daemon --foo “${FOO}”’ but if FOO isn’t set in the configuration file then we don’t want the --foo option appearing at all. We can manage this using one of the less well known shell substitution modifiers. The substitution ${FOO:+bar} has no value if FOO is not set and the value bar if FOO is set (regardless of FOO’s value). Further variable substitution can be done within bar. Therefore, the expression ${FOO:+--foo “${FOO}”} has no value if FOO is not set and the value --foo “${FOO}” if FOO is set. We will use this syntax wherever there is an optional argument.

The configuration file takes the form of many KEY=VALUE lines, and that can be written from scratch in the %build section simply by running cat with macro evaluation:

```
# orbd config file
cat <<EOF > %{name}-orbd.cfg
ORBDUSER=j2orbd
ORBDGROUP=j2orbd
LOGFILE=%{var}/log/%{name}-orbd.log
PIDFILE=%{localstatedir}/run/%{name}-orbd.pid
DBDIR=%{localstatedir}/%{name}-orbd
EOF
```

If we are setting variables in a configuration file then we need to test them. There should be tests for all variables before they are passed to the master daemon so that useful error messages can be generated and a daemon doomed to failure is never launched.
Creating a new user

• Trying to create a new group:
  groupadd -r j2orbd || true

• Creating a new user if necessary
  id j2orbd || useradd -c "Java 2 ORB daemon"
  -d %{_localstatedir}/%{name}-orbd -g j2orbd
  -s /bin/false -r j2orbd

• This does not create the home directory!
• There is no way to test for a group!

We must create a new user, with its own group, to run the daemon as. We do not want to run the daemon as root under any circumstances. If the daemon goes wrong or is compromised in an attack then we want the damage as restricted as possible. The Red Hat useradd program runs by default with the "user private group" scheme where every user has a group of their own. This excellent model has not been followed by SuSE so to implement it generally we will have to do it ourselves in parts.

First we try to create a group, because it must exist prior to being passed as an argument to the useradd command. There is no way to test for the existence of a group prior to trying to create it so we try to create it and catch any error code. Note that we cannot let the line return an error because that would terminate the scriptlet and the installation as a whole. The "-r" option causes this to be a "system group" and have an ID below the minimum ID allowed for real users (typically 500).

We can test for the existence of a user, with the id command. So, if the user does not exist then we create it. This is slightly cleaner than just throwing away error conditions like we did for groupadd. Note that the "-r" option on useradd creates a system user analogous to groupadd. With useradd it has the additional effect of stopping the creation of the home directory. The home directory for a system user is a nebulous concept at best, so I am using the database directory location, as we will have to create that anyway. By default useradd creates accounts with locked passwords (so it is impossible to log in as j2orbd) and to be on the safe side we are setting the shell to be /bin/false.
Before the installation

- Installing the script is not enough
- Actual program runs as a particular user
- We need to create a user at installation time
- User will own (at least) the database directory
- User must exist before the installation

We are going to run the orbd as a non-root user, j2orbd. This user will need to maintain (and therefore write to) the state database so we will need to have files installed by our package whose permissions refer to this user. Therefore the user must exist before the files are installed and our package needs to create it.

The RPM system has hooks to allow small scripts to be run before and after the installation (or deletion) of the files. These are introduced by the four commands:

- `%pre` Commands run before the installation of the files during `rpm --install` or `rpm --upgrade`
- `%post` Commands run after the installation of the files during `rpm --install` or `rpm --upgrade`
- `%preun` Commands run before the removal of files during `rpm --erase` or `rpm --upgrade` etc.
- `%postun` Commands run after the removal of files during `rpm --erase` or `rpm --upgrade` etc.

Most often, the `%postun` script undoes the work of the `%pre` script and the `%preun` script undoes the work of the `%post` script. This is not universally true and we will be meeting an exception here. We will create the j2orbd user in the `%pre` section but do not delete it in `%postun`. Some files may have been created by the service which are owned by that user. Therefore we will leave the user behind. There are (at least) two reasons for this. The first is that it is easier for the administrator to determine the origins of the file given the name j2orbd than some anonymous numeric user. The second is that the numeric userid might be reassigned later to another system account and this leads to “leakage” between two services that should be kept apart.

```
%pre orbd
# Make sure a system group "j2orbd" exists
groupadd -r j2orbd || true
# Now create a user
id j2orbd || useradd -c "Java 2 ORB daemon" -d %{_localstatedir}/%{name} -o
orbd -g j2orbd -s /bin/false -r j2orbd
```
Adding the database directory

- Configuration file line
  - DBDIR=%{_localstatedir}/%{name}-orbd
  - Create the directory in the build root
- Needs to be owned by the j2orbd user
- Use %attr(02770,j2orbd,j2orbd)
  - Overrides %defattr
- Use %dir

We have wired in the location of the database directory in our configuration file and set it as our user's home directory without creating it with the useradd program. We must create the directory in our build root to put it in the package. This directory must be writable by the j2orbd user which now exists following its creation in the %pre script. But because we are not building our package as root and because the j2orbd user may not exist on our build host we cannot change the directory to that user. Instead, we specify the mode, user and group of the directory with the %attr directive which overrides the default attributes given by %defattr.

Note the choice of mode. Obviously we must make the file writable by the j2orbd user, but we also make it writable by the j2orbd group. This allows us a little extra flexibility in case we have some ORB administrators who may need to intervene manually from time to time. To make sure that all groups remain j2orbd we set the setgid bit on the directory's mode.

Also, because we are adding an empty directory to the package rather than identifying an empty directory tree of files to add we also qualify the line with the %dir flag.

%install
mkdir --parents ${RPM_BUILD_ROOT}%{_localstatedir}/%{name}-orbd
%files orbd
%attr(02770,j2orbd,j2orbd) %{_localstatedir}/%{name}-orbd
Logging

- Daemons (should!) generate logs
- Options:
  - Own logging
  - "System" logging
  - Standard output/error
- Description in %files

Every system service should log its use and the Java 2 ORB service is no exception. Unfortunately this is where we see just how poorly written the service is. A daemon can log in one of two ways. The first is for it to manage its own logging. One of its configuration options should identify a log file and it should have facilities to reopen that log file according to some external stimulus, typically an operating system signal. A "poor man's" version of this is provided by using the system's own logging facility, "syslog". Unfortunately the Java 2 ORB daemon does neither: it simply logs to standard output and this needs to be manually redirected.

But what should a log file look like in the %files section? It doesn't really exist until the daemon starts to run (and has something to log) and we never launch the daemon as part of making the package. What we do is to add a "ghost" entry in the %files section. Quite literally, the attribute is "%ghost". This tells rpm not to install the file listed when the package is installed but should it come into existence later then it belongs to the package. Paradoxically, the file does have to exist in the build root, even though it won't be installed.

%install
  mkdir --parents %{RPM_BUILD_ROOT}%.var/log
  touch %{RPM_BUILD_ROOT}%.var/log/%{name}-orbd.log
%files orbd
  %ghost %{var}/log/%{name}-orbd.log

We should address the difference between the "missingok" configuration files and the "ghost" logging files. Files flagged as "%config(missingok)" are installed by rpm but package verification should not be too concerned if they are deleted by the system administrator later. Files flagged with "%ghost" aren't installed at all but are claimed if the system administrator later creates one manually.
Log rotation

- Logs must be rotated
- logrotate facility
- /etc/logrotate.d/
- Problems with standard output
  - Restart the daemon
  - “Postrotate” script

Now, let’s see why logging to standard output is such a problem. Log files grow; it’s in their nature. Therefore some means to control their growth is required. The technique used is called “log rotation”. A simple example would run like this: at regular intervals if the file logfile.2 exists it is deleted, if the file logfile.1 exists it is renamed to logfile.2, if the file logfile exists it is renamed to logfile.1 and a new, empty, logfile is created. Something is then done to cause the daemon to log to logfile rather than logfile.1 (its original log file). It is in the “something must be done” that the problems arise. Daemons that run their own log files, including the central syslog daemon are designed so that when they receive a particular signal they close their log files (logfile.1 by now in our example) and reopen them by name (logfile in our example). So, if the signal is sent immediately after log file rotation everything is sweet. However, some services are written simply to log to standard output which must be logged somehow. We redirect the output to append to a file either explicitly in the case of the Red Hat script or implicitly in the SuSE script. But there is no scope for reopening the log file short of shutting down and restarting the service.

Log file rotation is managed from files in the /etc/logrotate.d directory. Typically there is one per package that generates logs and these are regarded as configuration files, suitable for system administrators to change. Because it has to refer to the location of the log file (which uses %{_var}) and call the startup script (%{_initrddir}) we will create it from within the spec file to use the macro facility in the %build section.
(We could use sed from a template source file if we wanted.)

%install

# orbd log rotation
cat <<EOF > %{name}-orbd-logrotate
%{var}/log/%{name}-orbd.log {
  daily
  rotate 14
  missingok
  create 644 j2orbd j2orbd
  Postrotate
    %{initrddir}/%{name}-orbd restart
  endscript
}
EOF
%files orbd
%config %{sysconfdir}/logrotate.d/%{name}-orbd
After the installation

- Installing the script is not enough
- Set of symbolic links to it to manage service
  - We need to run chkconfig at installation time
  - chkconfig --add
  - Installation vs. upgrade?

Even after the right files have been installed on the system our work is not done. The startup script itself is not enough. Linux determines what services to launch at boot time by a set of symbolic links pointing to this script. These scripts need to be created and that is done by the chkconfig program. If we have installed a script as `%(initrddir)/j2sdk-orbd` then the command “chkconfig --add j2sdk-orbd” will set up the symbolic links. However, if this is an upgrade then we should not trample on any changes to the symbolic links made by the administrator. So, we need to run a program after the installation of the files (i.e. in %post) and it needs to know whether the files were installed as an initial installation or as an upgrade.

Each of the pre-/post-installation scriptlets is called by rpm with a single, numerical argument which is the number of instances of the package that will exist after the current rpm activity is run. For these purposes, upgrades consist of an installation of the new version as one “rpm activity” followed by the erasure of the old version (the second activity). Therefore while the upgrade scripts for the new version are being run the number of instances of the package is two. This is best demonstrated with an example. Suppose we have a dummy package that does nothing except run all four scripts, each of which is defined like this:

```
%post
echo Post-install: Version=%{version} Arg=$1
```

then we get to see the following behaviour:

```
# rpm --install dummy-1-1.i386.rpm
Pre-install: Version=1 Arg=1
Post-install: Version=1 Arg=1

# rpm --upgrade dummy-2-1.i386.rpm
Pre-install: Version=2 Arg=2
Post-install: Version=2 Arg=2
Pre-uninstall: Version=1 Arg=1
Post-uninstall: Version=1 Arg=1

# rpm --erase dummy
Pre-uninstall: Version=2 Arg=0
Post-uninstall: Version=2 Arg=0
```

So, for the %post script, if the argument is 1 we should add the service. In the %preun script, if the argument is 0 we should remove it.
Running chkconfig

- `%post`
  - Post-install
  - One instance = fresh install
  - Two instances = upgrade
- `%preun`
  - Pre-uninstallation
  - Zero instances = erasure
  - One instance = upgrade

So, in the `%post` script we need to check the (single) argument to the script to see if it is 1. If it is then this is a fresh installation and we need to add the service with “chkconfig --add”. If the number is 2 or greater then the service already exists and we should leave the runlevel configuration alone.

Similarly, in the `%preun` script we check the argument to see if it is 0. If it is then we are removing the package entirely and we should delete the service with “chkconfig --del”. If it 1 or more then the service will exist after this operation and we should leave it alone.

```bash
%post orbd
if [ ${1} -eq 1 ]
then
  chkconfig --add %{name}-orbd
fi

%preun orbd
if [ ${1} -eq 0 ]
then
  chkconfig --del %{name}-orbd
fi
```
Release 4

- Build the packages!
  - j2sdk
  - j2sdk-demo
  - j2sdk-src
  - j2sdk-path
  - j2sdk-orbd

And that's it! We can build the packages and all's well.