Do-It-Yourself RPM Packages

Private Packaging Service

Most Linux software is available as source code and needs to be compiled. This may be a simple task, but problems start to occur when you need to remove an application.

BY ANDREA MÜLLER

Isn't the amount of self-compiled software that accumulates on a machine over a length of time amazing? Although you might know where those binaries got to, the */usr/local/lib* subdirectory tends to put you at a total loss. Fortunately, you can avoid manual deletion and puzzling over the origin of a file if your distribution's package manager also handles self-compiled software.

The most popular program for this task is *rpm*, the Red Hat Package Manager [5]. It not only stores information on the installed packages in its database, but also records their dependencies to allow for clean removal. This is an important benefit because, although you may have installed a library in order to run program *A*, you cannot simply remove it along with *A*. Doing so might affect other programs installed subsequently that also rely on the library in question. RPM monitors the attempt and warns you if this is the case.

Manual Labor Preferred

To allow RPM to manage self-compiled software, the source code has to be packaged first. The *checkinstall* [1,2] program monitors the installation and creates an RPM package from the files copied when you call ***make install***.

Like any automated solution there are a few drawbacks, such as the fact that it does not support all the capabilities the RPM command provides. *checkinstall* cannot distribute a software program across multiple packages and even refuses to cooperate entirely in some cases.

The worst issue is the fact that the file list for package is created during the installation process, in other words, you have no way of knowing whether you will be overwriting existing files. However, this issue is on the shortlist for a future version which will support the RPM-*BUILDROOT* option. The option allows you to build packages in a kind of sandbox, in the */tmp* directory, for example. The program sees */tmp* as its root directory for the installation, thus avoiding any possibility of overwriting system files.

If you prefer to avoid half-hearted attempts, and additional tools, you might decide to build your own packages. Unfortunately, the RPM documentation does not provide any clues as to how to do this, and "Maximum RPM" [3], the reference for current and aspiring package builders is not recommended as a quick reference, as it weighs in at several hundred pages.

Instead of boring you to death with tons of theory, lets look at how to build a package for the *xpuyopuyo* [4] program, a network-aware and highly addictive Tetris clone with neat graphics and comic sounds (see Figure 1); that alone should make the effort of building the package worthwhile.

Preparatory Steps

If you think you can just launch into this, think again. Most distributors only install programs they consider absolutely necessary to install RPMs, but definitely not the tools you need to create an RPM. In our case this means we need to install the *rpm-build* package before we can start.

Our packaging counter, that is the working directory for the package builder, is located below */usr/src*, and again every distribution does its own thing. SuSE calls it *packages*, Mandrake prefers *RPM*,

GLOSSARY

make install: *This command runs the install section of the makefile, copying the files belonging to an application to the appropriate target directories on a system.*

and the US market leader opted for *redhat* in a flight of egocentricity.

The five subdirectories below this level, *SOURCES*, *SPECS*, *BUILD*, *RPMS*, and *SRPMS* are used for package building. *RPMS* and *SRPMS* are receptacles for storing the binary and source packages we will be creating, and the *BUILD* directory is used for compiling the software. As you cannot launch the commands required to create an RPM interactively – instead RPM parses a so-called specfile containing the commands – there is nothing for you to do in this folder.

The package builder uses the *SOURCES* and *SPECS* directories. You need to place the *tar.gz* archive with the software you will be packaging below *SOURCES*. If the source code is not available in *tar.gz* format, you will need to convert it first.

Assume superuser privileges by calling *su* and typing the *root* password. You will need superuser privileges throughout the build process. You can allow non-privileged users to build packages, but this is complicated to set up. Download the *http://chaos2.org/xpuyopuyo/xpuyopuyo-0.9.8.tar.gz* file and copy it to *SOURCES*.

Then change to the *SPECS* directory. This is where RPM will expect the specfile. There are no restrictions on the file name, although a naming convention along the lines of *programname.spec* is common. While still working with *root* privileges, open an editor and create the *xpuyopuyo.spec* text file.

The Preamble

A specfile comprises three sections. The first section, known as the preamble, contains generic information which will later be used to create the package name,

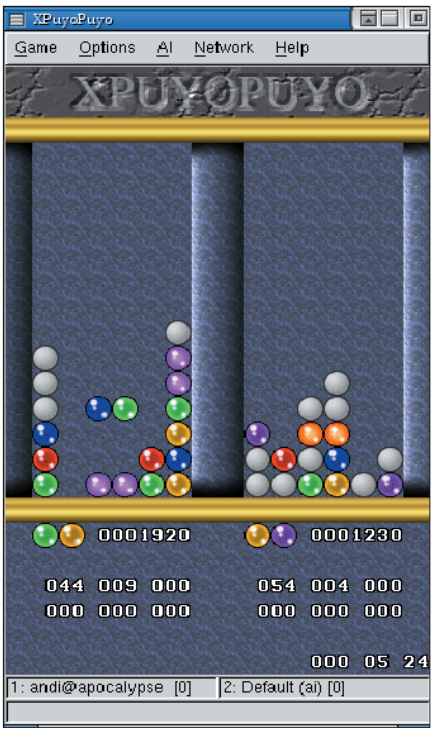

Figure 1: Xpuyopuyo – our sample program used to demonstrate building an RPM package

for example. The preamble for *xpuyopuyo* might appear as shown in Listing 1.

The structure is extremely simple, and so similar for each package that it makes sense to work with templates: an RPM specific tag followed by a colon and the corresponding value. The values for *Name*, *Version*, and *Release* will later form the package name. The *Version* tag must not contain certain characters like a space or minus. Some other characters may cause problems to, so only use alphanumerical characters and dots to be on the safe side.

The *Release* tag is a kind of version number for the package. Distributors often compile multiple packages for the same program version, possibly because patches have been released, or the distributor has enhanced the program in some ways. This field should be used to add a distribution specific ID, but only Mandrake uses it consistently in this way. Mandrake package names will always contain the *mdk* string.

Of course, you can add what ever you like – we will be using the hostname in our example. You should note that some characters, like the minus sign for example, are not permissible. The *Summary* field provides space for a short description, the *Source* tag stores the name of the sourcecode archive and the *URL* field provides service information for package users. Users can tell at a glance where the software in the RPM comes from.

The *Distribution* line is optional – you can omit it if you like. However, add your name, or nick, and a valid email address after the *Packager* keyword, that is, if you intend to make your do-it-yourself package available to other users. It is easy to make mistakes when building packages, and users will be happy to have someone to help them out, should this happen.

Let's take a quick look at the *Group* item. This is where you specify the group that GUI-based package managers, like *kpackage* for example, should categorize the software. You could create your own groups in theory, but if you intend to distribute the packages you build, you should try to stick to the current naming conventions. Check the RPM documentation in the *GROUPS* file below */usr/share/doc* for the current groups.

The Building Plan

The main section of the specfile contains the instructions for unpacking, configur-

Listing 1: Specfile preamble for the Xpuyopuyo package

```
#Specfile for xpuyopuyo
Summary: Network-aware Tetris Clone
Name: xpuyopuyo
Version: 0.9.8
Release: apocalypse
Copyright: GPL
Group: Games/Arcade
Source: xpuyopuyo-0.9.8.tar.gz
URL: http://chaos2.org/
Distribution: Mandrake 9.0
Packager: Andrea Mueller <amueller@linux-user.de>
```

Listing 2: Main section of specfile for Xpuyopuyo

```
%description
Network-aware Tetris Clone with cool sound.
%prep
%setup
./configure --prefix=/usr/local --with-gnome=no
%build
make
%install
make install-strip
cp doc/xpuyopuyo.txt /usr/local/share/xpuyopuyo
```

ing, and installing the software. The procedure can differ from one application to the next, although the three command trick is typical

```
configure
make
make install
```

This is also true of *xpuyopuyo*. Just add the lines from Listing 2 to your specfile.

RPM-specific keywords are followed by percent signs. The *%description* item contains the text that appears when a user types *rpm -qi xpuyopuyo* to display information on the package. This is a multi line entry that can contain any character types. The *%prep* section terminates the description and initiates the build process preparation tasks. This phase takes care of any steps needed to get the source package ready for compiling.

The *%setup* item introduces a simplification that RPM provides for package builders: the use of macros. Just like Office programs can record macros to simplify recurring tasks, RPM has similar tools. The Setup macro is probably the most popular, and takes care of several tasks. It unpacks the archive file stored below *SOURCES* in *BUILD* and then uses the *cd* command to change to that directory. If it discovers a previous attempt to build the current package, the Setup macro will remove any residues created below *BUILD* by the previous compilation attempt before unpacking the sources.

The next line in the *%prep* section launches the actual build. You would typically call the *./configure* command to get the program ready for compiling. The *--prefix* parameter is just used as an example in this case, as the script will default to this directory as the target directory for the program. This is where you add the *configure* options that control the configuration script in the shell. The package should be installed in */usr/local* just like any other self-compiled package. This allows you to segregate self-compiled software and distribution packages. It also means that updates should be no problem, provided your distributor maintains its packages carefully.

Listing 3: File list for Xpuyopuyo specfile

```
%files
/usr/local/man/man6/xpuyopuyo.6
/usr/local/bin/xpuyopuyo
/usr/local/share/xpuyopuyo
%doc /usr/local/share/xpuyopuyo/xpuyopuyo.txt
```

The *configure* option, *--with-gnome=no*, which is typical of *xpuyopuyo* prevents GNOME desktop icons from being created and copied to the directories below */usr/share*.

RPM actually compiles the software in the *%build* section that now follows. The *make* command does this in *xpuyopuyo*'s case, just like the majority of packages. The next command is, of course, *%install* – in our example the *make install-**strip*** command is called to install the compiled software in */usr/local*.

This completes the main section of the specfile, although there are other sections that can be added here, such as the *%patch* macro that automates patch application. (Patches are used to correct errors or provide additional functionality to the software.) The *%package* instruction divides the software up into multiple packages, such as *program.rpm*, *program-doc.rpm* and *program-devel.rpm*. If you build a package that contains a library, you should add the following lines to your specfile:

```
%post
ldconfig
```

This calls the *ldconfig* command after installing the package, and registers the newly installed library, allowing other programs to discover it.

You can, theoretically, run arbitrary commands in the subsections of the main section of a specfile. However, you should restrict the sections that will run later when the package is installed (an example would be the *%post* section) to commands that are guaranteed to be available on any Linux system.

GLOSSARY

strip: *This command removes symbols from binary files, thus reducing their footprint. Symbols are used by developers when debugging a program, and can be output using the nm command. The output can show the functionality provided by a library, for example. Removing symbols from a binary file will reduce its footprint considerably, and this is why most distributors provide only stripped programs and libraries.*

The Hard Part

Everything is set up, but RPM still does not know which files belong to the package. In fact, you don't know this either, as you have never installed *xpuyopuyo* previously. There is no easy way to create a file list. Some people install the software up front, and use the *installwatch* program to create a list of the files copied during installation. This is exactly what *checkinstall* does, and the method is not 100 percent foolproof.

Hardliners will assemble the file list on the basis of the *Makefile*. As an alternative, you could install the software as a normal user, using a subdirectory of your home as the target. You could write a script to compile a file list, or redirect the output from *ls -R*. Of course, you will need to modify the paths in the specfile: instead of */home/username/test/bin/file*, the specfile will read */usr/local/bin/file*.

Although it might be tedious, the latter method is preferable. By installing the package as a normal user, you ensure that "*make install*" will only install to the subdirectories below the supplied *--prefix*, and you can ensure that the *Makefile* does not include a killer command like "*rm -rf*". Also, this allows you to check that the developer packages required by the build process are available (*xpuyopuyo* needs the following development packages: *gtk*, *xpm*, *XFree*, *glib*, and *mikmod*).

Some friendly programmers actually include a specfile, and a complete file list, with their software. Now this might sound like your favorite TV cooking program, but I got this ready while I was waiting: Listing 3 shows the last section of the specfile.

The last two lines are interesting. Multiple subdirectories with images and sounds are copied to */usr/local/share/xpuyopuyo* by the installation process. If the *%files* section includes a directory, the directory and its contents are packed and become part of the package – so

Figure 2: RPM has created binary and source packages

avoid adding the whole of */usr/local* in this section.

The *%doc* prefix in the last line, tags the file that follows as documentation. This file is displayed if you type *rpm -qd xpuyopuyo* to query the package.

The *%doc* tag also has another use: programs often supply documentation that *make install* does not copy. If the files section of the specfile contains a line such as

```
%doc README FAQ CREDITS
```

RPM will copy these three files from the source code directory to a subdirectory of *defaultdocdir* and add them to the package. The *defaultdocdir* differs depending on your distribution, defaulting to */usr/share/doc* on Mandrake and Red Hat, and to */usr/share/doc/packages* for SuSE. The default can be modified by editing the *.rpmrc* file in the home directory for *root*.

Wildcards in the *%files* section of the specfile can make life easier. If a program copies a whole bunch of localization files to */usr/local/share/locale*, you can use a single line to include all of them:

```
/usr/local/share/locale/*➥
/LC_MESSAGES/programname.mo
```

RPM will complain if some of these language directories do not contain a file matching this pattern, although execution will continue normally.

Building the Package

We are finally ready to build the package. To do so, call the *rpm -ba xpuyopuyo.spec* command in the *SPECS* directory; Red Hat users will need *rpmbuild -ba xpuyopuyo.spec* instead. The *-b* parameter tells RPM to parse a file to obtain control information for the build process. In this case we are looking to perform a complete build. RPM should work its way through the specfile sections and create both a binary and a source package, as is indicated by the *a* parameter.

If you want to omit the source RPM, choose *b* instead, that is, you type *rpm -bb xpuyopuyo.spec*. The *rpm -bc xpuyopuyo.spec* tests whether the package will compile, running the specfile up to the *%build* item, that is without copying any files.

No matter what command you issue, *rpm* is extremely verbose. You can see what the package manager is doing and are also shown the output created by the individual commands:

```
+ cd xpuyopuyo-0.9.5
+ ./configure -prefix=➥
/usr/local
creating cache ./config.cache
```

Lines starting with plus signs show what RPM is currently doing; in this case, changing to the source code directory and calling the *./configure* command, followed by the output from the script. If everything works out okay, the last few lines should appear as those shown in Figure 2.

Before the success message, you can see that RPM to some extent attempts to ascertain the package's dependencies. To do so, it calls the *ldd* command for each binary file to check the libraries the file is linked against. The *xpuyopuyo* package will refuse to install if the *mikmod* library is not found, for example. Dependencies that *ldd* does not recognize must be specified using the optional *Requires:* tag in the preamble.

The mail program, Mutt, needs a local mail server, for example, that is it needs a tag that reads *Requires: mailserver*. The counterpart to this tag is the *Provides:* tag. Distributors who maintain their packages carefully will add *Provides: mailserver* to the preamble for *sendmail* and *postfix*.

The complete binary package is now stored in the *i686* subdirectory below *RPM*. Each distribution optimizes for a different processor type, but you can overwrite the default using the *rpm* command line switch *--target processortype*.

The "*make install-strip*" command in the specfile copies the *xpuyopuyo*, but it does not add the newly created package to the package database. To add the Tetris clone to the database, you will need to install the package using *rpm -i ../RPMS/i686/xpuyopuyo-0.9.5-apocalypse.i686.rpm*. If you are not in the *SPECS* directory, supply the absolute pathname. You can then use the *rpm -qil xpuyopuyo | less* command to query your do-it-yourself package for information.

Most of you will at some time have installed a package that did not turn out as expected. This may be caused by a single missing file. To save your users from this kind of frustrating experience, ensure that you test your do-it-yourself RPMs on another computer – not the build system – before you distribute them. ■

INFO

[1] Checkinstall: *http://asic-linux.com.mx/~izto/checkinstall/*

[2] Christian Perle: "Say Hello, Wave Goodbye", Linux Magazine Issue 22, July/August 2002, *http://www.linux-magazine.com/issue/22/checkinstall.pdf*

[3] Maximum RPM: *http://www.redhat.com/docs/books/max-rpm/*

[4] Xpuyopuyo: *http://chaos2.org/*

[5] Search engine for Linux rpm packages: *http://www.rpmseek.com*