

# Package installation made easy UNWRAPPING THE PACKAGE

Unlike their Windows counterparts, Linux programs rarely come complete with a setup program that checks your disk for free space, creates an installation folder and sets up icons for you to run the program. Under Linux this must all be done manually. But don't panic: Hans Georg Esser show you it's not as hard as it sounds.

---

**Binary:** Binary files, often just called »binaries«, are programs that have been compiled into a form that can only be understood by the computer. They are kept in directories such as `/bin`, `/usr/bin` and so on.

**Source:** Source files are the original text files containing the program statements (in a language such as C or C++) that the programmers wrote. You can read them, though unless you're a programmer too you might not understand them!

---

There are many different types of software archive in the Linux environment. Before we look at how to install them, let's find out a bit more about them.

## rpm packages

If you're already running Linux you'll no doubt have already found many files with names ending in `.rpm`. This stands for Red Hat Package Manager and identifies archives which have been compiled according to a standard introduced by Red Hat. Nowadays this format is used by almost all distributions.

But there's an important difference between **binary** RPMs and **source** RPMs. Binary RPM packages contain the executable files as well as configuration and miscellaneous other files that go together to form the application. In addition to this, a binary RPM archive holds information on what to do immediately before and after the installation. It also works out if any other packages are required for the installation and if so, if any file conflicts might occur.

Source RPMs contain the program source code (the text written by the programmer, usually in C or C++), together with instructions showing how to compile this code into something useful. From a source RPM you can produce a binary RPM package. More on this later.

In most cases, RPM packages can be installed either by typing a command into a terminal window or by using a graphical tool under KDE or Gnome. In order to do this, system administrator (`root`) rights are required. So before starting you must login as user `root` or use the `su` command. Owing to differences in individual Linux distributions it's often the case that a particular RPM package can only be installed on the distribution for which it was created. Therefore, when searching for files on the Internet you'll frequently find different RPM packages for different distributions.

The filename lets you recognise the version of the program and the platform for which it was created. A typical example might be:

```
kpackage 1.3.10 3.i386.rpm
```

The version number here is 1.3.10. The number 3 after this means that the package has been created three times, implying there are packages num-

bered 1.3.10-1 and 1.3.10-2 also around which perhaps were made during an earlier version of the distribution. »i386« indicates that the program will run on all Intel based systems (all computers with 80386, 80486, Pentium I/Pro/III/III or compatible processor, including AMD and Cyrix etc.).

If you find several versions of a program you're interested in but each has a different ending like `i386.rpm`, `i486.rpm` or `i586.rpm`, select the one that best suits your computer. Packages which were compiled for Pentium (586) processors are better optimised than 386 packages since they use additional commands which the 80386 lacks.

Source RPM packages have the abbreviation »src« in the name instead of the platform designation - the source package for the above RPM package might therefore be called:

```
kpackage 1.3.10.src.rpm
```

Source RPM packages don't need a platform indication because under Linux the source code is - in general - not specific to a hardware platform. It is the process of compiling to a binary file, translating the source code into machine language, which makes a program platform-specific.

## deb packages

In addition to RPM there is another popular package format: the Debian format. Debian packages end in `.deb` and are used by the new Corel Linux distribution as well as Debian's own distribution. Owing to the relatively limited use of Debian - because Debian is one of the less user friendly distributions - we will dispense with a detailed description. It is worth noting that Debian and RPM packages can be converted in one another with the aid of the program *alien*, though this has a variable success rate. If possible you should choose a package type that suits your system.

## tar.gz archives

Files that end in `.tar.gz` are broadly equivalent to *zip* archives under Windows. But while *zip* archives frequently compress the contents of a whole folder and subfolders in one go, this is a two step proce-

ture with Linux. First, a *tar* archive is created which contains the folder hierarchy. The files are not compressed until the second step is instigated, in which the program *gzip* is used to pack the *tar* archive. This two stage process explains the double file extension (ie *package.tar.gz*.)

Program packages in *tar.gz* format usually contain the program source. For installation to take place they must be unpacked, configured, compiled (turned into a binary program) and finally copied to the correct place in the Linux folder hierarchy. We'll describe in more detail how this happens later on. Occasionally you will find *tar.gz* packages which contain compiled program files but it's pretty unusual.

## tar.bz2 archives

These are a variant of *tar.gz* archives. After *tar* has run the compression program *bzip2* is used which achieves a higher compression rate than the older *gzip* program. However, the archive doesn't differ significantly from the *tar.gz* archive apart from the fact that a different command is necessary to unpack it.

Now that the overview is complete we can discuss the installation procedures. We'll start with the simplest variant: the installation of RPM archives.

## Installing RPMs

Once you've found a binary RPM package that suits your Linux distribution you can install it via the console using the *rpm* command or via a graphic front-end like *kpackage* under KDE or *gnorpm* under GNOME. Some distributions supply further tools which fulfil the same purpose, such as SuSE's YaST, easyLinux's eProfile or Mandrake's *rpm-drake*. If your distribution has its own installer it is better to use it as it may provide extra benefits like automatic menu configuration.

## kpackage

In order to install packages you must have administrator rights. Open a console window and type the

command *su*. You will be asked for the password for the administrator root. Once this is completed you can start the package manager by typing *kpackage*. Some distributions allow you to call up *kpackage* using the K menu without becoming root beforehand. When the program starts a window opens into which you must type the root password (Fig. 1).

When it starts, *kpackage* first reads the information in the RPM database which tells it about the program packages that are already installed. It then displays this in a tree view (Fig. 2.) Each package you've installed should be present in this hierarchy – for example, you would find the editor *emacs* under RPM/Applications/Editors.

In order to install a new package select the menu item File, Open. The usual Open dialog should appear and you will be able to select the package you want to install. *kpackage* will now display information on the package in the right half of the screen. Under the Properties tab you will find a brief description which amongst other things displays the name and version (Fig 3). Clicking on the tab File List should give you a listing of all the files which will be created during installation. You can find out from this what folders the files will be put in.

On the left you will find five check boxes: Upgrade, Replace Files, Replace Packages, Check Dependencies and Test. These have the following meanings:

- **Upgrade:** If you want to install a program that already exists on your system in an older version you must check this box to carry out an update. The older version is automatically uninstalled. If Upgrade is not checked and an older version is present, the installation will halt with an error.
- **Replace Files:** *rpm* keeps an eye open for files that already exist being overwritten by the installation. If so it halts with an error. This can happen when two packages place the same configuration file in the folder */etc*, for example. If you mark this field, a package is installed even if it means overwriting files that already exist.
- **Replace Packages:** This option is similar to *Upgrade*: a package is installed when it is already present in

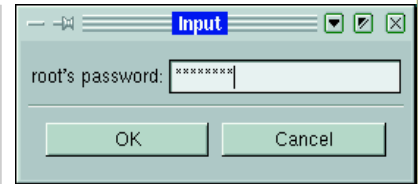
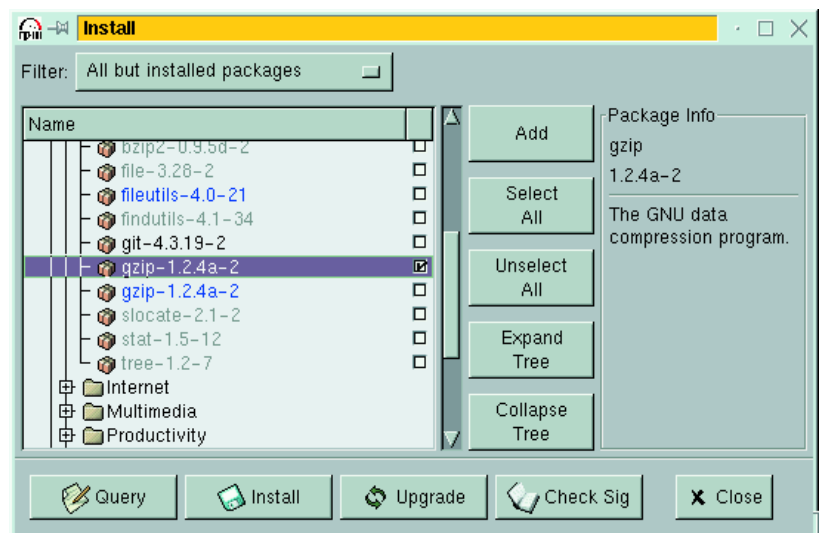
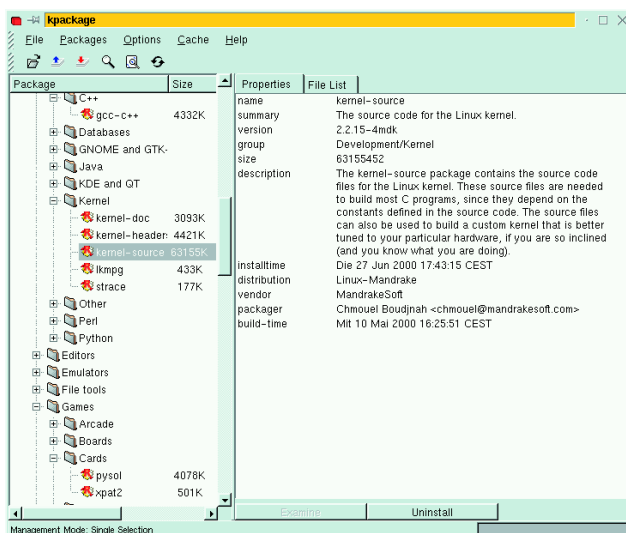
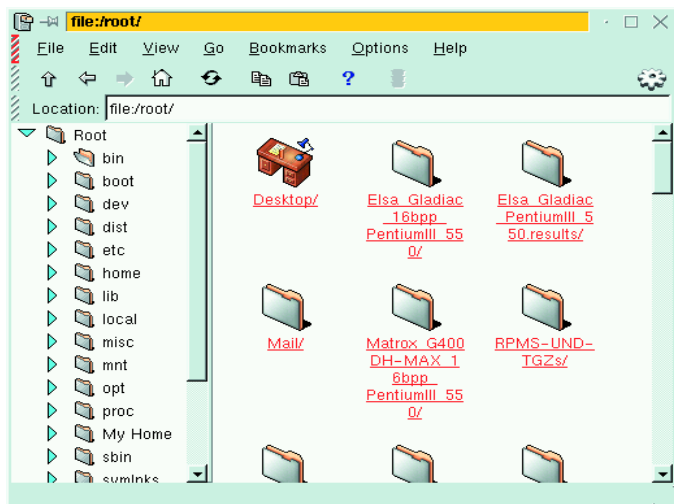


Fig. 1: When starting *kpackage* as a normal user you are prompted for the root password.

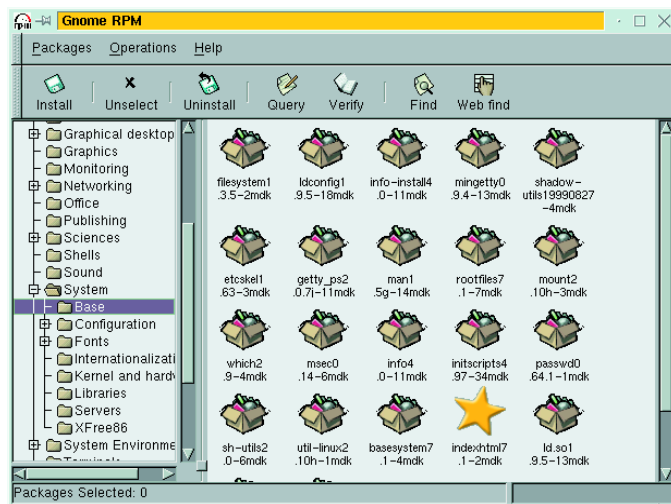
[below left]  
Fig. 2: In the tree view you will find all the RPM packages already installed.

[below]  
Fig. 3: Package properties using *kpackage*





**Fig. 4:** A red mark at top left means that this window has administrator rights



**Fig. 5:** gnorpm first displays all installed files

another version but the old package is preserved. This might be useful if perhaps you want to install two versions of a particular library file.

- **Check Dependencies:** As already mentioned, RPM packages “know” which extra packages are required by the one you are installing. An example of this is the KDE base package *kdebase* whose programs can only run if *qt* and *kdelibs* are also installed. If a required package is missing you will receive an error message. If you know for sure that all necessary files are present, you can uncheck this option. You might do this if for example you want to install a package designed for a foreign Linux distribution and you know that the required package has a different name and is not being found despite being present.
- **Test:** This is simple enough: it checks whether the package can be installed without difficulty. Placing an check in this field means that despite going through the motions, no files will actually be installed.

After making any changes to these settings click on *Install* to install the selected package. *Cancel* will return you to the tree view.

Instead of selecting a package using *File, Open* you can also drag it from a kfm window to the kpackage window. If kpackage is not yet open you can start kpackage from kfm by clicking on the rpm archive. This needs you to be user *root*, however. There is a way to call up kpackage with the necessary rights from the kfm window: select the menu option System, KFM file manager (Super User Mode). After entering the password a kfm window is opened in which you have administrator rights: you can tell because of the red mark at the top left of the kfm window. If you click on an rpm archive in this window, kpackage is started without a password needing to be entered.

### gnorpm

Gnome has a similar program: gnorpm. This is superior to Red Hat's own tool glint and has the benefit of being available in all Linux distributions.

In principle, the same thing is done here as with kpackage. Start the program as the administrator root (type »su« and enter the administrator password). You'll also have to open a gmc (GNOME Midnight Commander) file manager window and drag the RPM package from it to the gnorpm window. This should open a new window, *Install*, in which the package is displayed (Fig 6). By clicking on *Queries* you can obtain more detailed information (the current gnorpm version 0.9, however, always crashed when we tried it.) A click on *Install* starts the whole thing running.

gnorpm also checks for package conflicts, and, if applicable, pops up a warning dialog box in which you can alter the installation settings.

### Installing RPMs by hand

In addition to using a graphical front end which simply calls the rpm routine, it is also possible to manually use rpm within a console. If you are not afraid of a hands-on approach then you will find this method more efficient.

All you need before starting is the precise file-name of the RPM, including its path (if the archive is not in the current folder.) You must then become the administrator root using su. Once you have done this, enter the command:

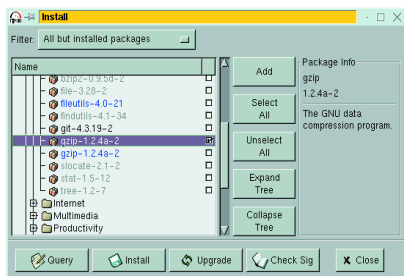
```
rpm -Uvh path/package 1.2.3 1.i386.rpm
```

It's as simple as that! During installation a progress meter is displayed showing how much of the work is completed. rpm can work on several packages at the same time, using a command like:

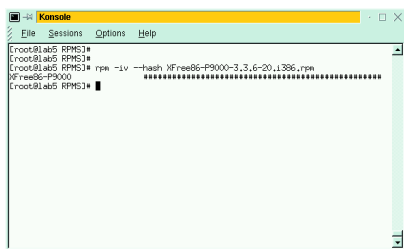
```
rpm -Uvh download/*.rpm
```

You can find out which version of a package is installed by typing rpm -Uvh q package name (see Fig. 7). You will find a list of the options that can be used in Table 1.

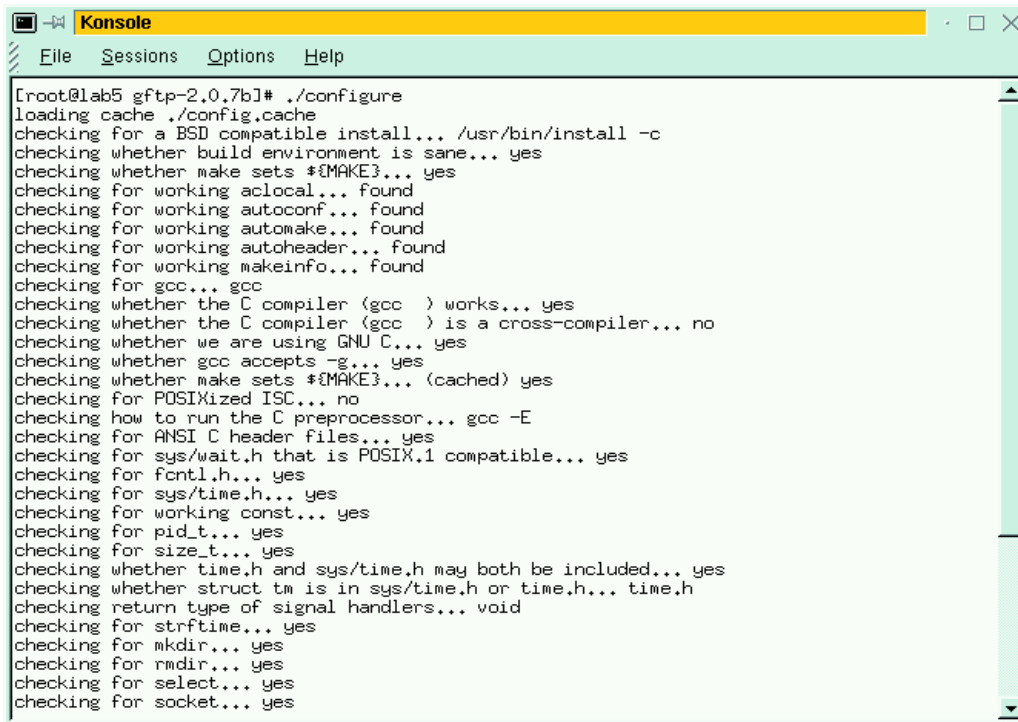
When using the options -q and -e for querying or erasing, only the package name (without the version number) needs to be indicated, i.e. not rpm -e package 1.2.3.rpm but simply rpm -e package.



**Fig. 6:** Clicking on Install starts the whole thing going



**Fig. 7:** Package installation and querying using a terminal window



```

[root@lab5 gftp-2,0,7b]# ./configure
loading cache ./config.cache
checking for a BSD compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking whether make sets ${MAKE3}... yes
checking for working aclocal... found
checking for working autoconf... found
checking for working automake... found
checking for working autoheader... found
checking for working makeinfo... found
checking for gcc... gcc
checking whether the C compiler (gcc ) works... yes
checking whether the C compiler (gcc ) is a cross-compiler... no
checking whether we are using GNU C... yes
checking whether gcc accepts -g... yes
checking whether make sets ${MAKE3}... (cached) yes
checking for POSIXized ISC... no
checking how to run the C preprocessor... gcc -E
checking for ANSI C header files... yes
checking for sys/wait.h that is POSIX.1 compatible... yes
checking for fcntl.h... yes
checking for sys/time.h... yes
checking for working const... yes
checking for pid_t... yes
checking for size_t... yes
checking whether time.h and sys/time.h may both be included... yes
checking whether struct tm is in sys/time.h or time.h... time.h
checking return type of signal handlers... void
checking for strftime... yes
checking for mkdir... yes
checking for rmdir... yes
checking for select... yes
checking for socket... yes

```

Fig. 8: ./configure : the first step to a finished program

## Compiling source files

Now we come to a more complicated way to install software – from the programmer's source files. Here the program exists in its basic form as a source code archive. Before anything else happens it must be unpacked to a suitable place: `/usr/local/src/` is usually the default. In order to do this you must become administrator root. You can then unpack the program archive. Archives which end in `.tar.gz` or `.tgz` are unpacked using:

```
tar xzf path/package.tar.gz
```

and packages which end in `.tar.bz2` are unpacked using

```
tar xIf path/package.tar.bz2
```

(Note: between `»x«` and `»f«` in the command above is a capital `»i«`.) This command creates a new subdirectory with the name of the source code files. You must now change to this directory. There then follows what some call the classical installation triple step: `»configure/make/make install«`:

```

[root@dual myprog 1.1.0]# ./configure
...
[root@dual myprog 1.1.0]# make
...
[root@dual myprog 1.1.0]# make install

```

All three commands will cause your screen to be filled with many system messages. What do they all mean?

Well, the first step `./configure` (which must be typed with a dot and slash before the word `»config-ure«`) starts a shell script in the current folder. This script has been created by the programmer and looks around your Linux system. It checks what operating system and what version you are using (frequently the same source text archive can be used on other Unix variants), which compiler is installed

(under Linux it's usually GNU C), and whether all the necessary program libraries exist in sufficiently up-to-date versions. If everything appears satisfactory the script produces a *makefile* (Fig. 8).

You need the makefile for the next two steps. When you run the program `make` (which must also be on your hard disk, of course), it processes the freshly created makefile which itself contains a recipe-like listing of what must happen – and in what sequence – in order to create a finished program. `./configure` and `make` can take quite a long time to run depending on the size of the program.

Finally, by typing `make install`, all created files will be copied to the correct places on your system. Programs themselves usually end up in `/usr/bin` or `/usr/local/bin`, help pages (man pages) in `/usr/man` or `/usr/local/man`, configuration files in `/etc` and so on.

Once this is all done, the installation is finished. Try running the newly-installed program. If it works properly you can delete the folder from which you carried out the compilation.

If after unpacking a source code package you find there is no `configure` file, examine the other files in the folder. Usually, you will find a `COMPILE` or `README` file in which the procedure for installing the program is described. ■

Table 1: rpm options

i	Install (no update)
U	Update
v	»verbose« (i.e. detailed) — displays package name
H	displays progress meter
q	»query«; enquires whether a package is installed
e	»erase«; deletes a package
nodeps	ignore dependencies (i.e. install even if necessary packages are missing)
force	force installation in the case of conflicts