Building and Installing Software Packages for Linux

Table of Contents

Building and Installing Software Packages for Linux

Mendel Cooper −−− http://personal.riverusers.com/~thegrendel/

v1.91, 27 July 1999

This is a comprehensive guide to building and installing "generic" UNIX software distributions under Linux. Additionally, there is some coverage of "rpm" and "deb" pre−packaged binaries.

7.Troubleshooting

8.Final Steps

9.First Example: Xscrabble

10.Second Example: Xloadimage

11.Third Example: Fortune

12.Fourth Example: Hearts

13.Fifth Example: XmDipmon

14.Where to Find Source Archives

15.Final Words

16.References and Further Reading

17.Credits

1.Introduction

Many software packages for the various flavors of UNIX and Linux come as compressed archives of source files. The same package may be "built" to run on different target machines, and this saves the author of the software from having to produce multiple versions. A single distribution of a software package may thus end up running, in various incarnations, on an Intel box, a DEC Alpha, a RISC workstation, or even a mainframe. Unfortunately, this puts the responsibility of actually "building" and installing the software on the end user, the de facto "system administrator", the fellow sitting at the keyboard –– you. Take heart, though, the process is not nearly as terrifying or mysterious as it seems, as this guide will demonstrate.

2.Unpacking the Files

You have downloaded or otherwise acquired a software package. Most likely it is archived (*tarred*) and compressed (*gzipped*), in `.tar.gz` or `.tgz` form (familiarly known as a "tarball"). First copy it to a working directory. Then *untar* and *gunzip* it. The appropriate command for this is **tar xzvf *filename***, where *filename* is the name of the software file, of course. The de–archiving process will usually install the appropriate files in subdirectories it will create. Note that if the package name has a *.Z* suffix, then the above procedure will serve just as well, though running **uncompress**, followed by a **tar xvf** also works. You may preview this process by a **tar tzvf filename**, which lists the files in the archive without actually unpacking them.

The above method of unpacking "tarballs" is equivalent to either of the following:

- gzip –cd filename | tar xvf –
- gunzip –c filename | tar xvf –

(The '–' causes the *tar* command to take its input from `stdin`.)

Source files in the new *bzip2* (`.bz2`) format can be unarchived by a **bzip2 –cd filename | tar xvf –**, or, more simply by a **tar xyvf filename**, assuming that `tar` has been appropriately patched (refer to the Bzip2 HOWTO for details). Debian Linux uses a different patch for `tar`, one written by Hiroshi Takekawa, so that the *–I, ––bzip2, ––bunzip2* options work with that particular `tar` version.

[Many thanks to R. Brock Lynn and Fabrizio Stefani for corrections and updates on the above information.]

Sometimes the archived file must be untarred and installed from the user's home directory, or perhaps in a certain other directory, such as `/`, `/usr/src`, or `/opt`, as specified in the package's config info. Should you get an error message attempting to untar it, this may be the reason. Read the package docs, especially the

README and/or Install files, if present, and edit the config files and/or Makefiles as necessary, consistent with the installation instructions. Note that you would **not** ordinarily alter the Imake file, since this could have unforseen consequences. Most software packages permit automating this process by running **make install** to emplace the binaries in the appropriate system areas.

- You might encounter shar files, or *shell archives*, especially in the source code newsgroups on the Internet. These remain in use because they are readable to humans, and this permits newsgroup moderators to sort through them and reject unsuitable ones. They may be unpacked by the **unshar filename.shar** command. Otherwise the procedure for dealing with them is the same as for "tarballs".

- Some source archives have been processed using nonstandard DOS, Mac, or even Amiga compression utilities such *zip*, *arc*, *lha*, *arj*, *zoo*, *rar*, and *shk*. Fortunately, Sunsite and other places have Linux uncompression utilities that can deal with most or all of these.

Occasionally, you may need to update or incorporate bug fixes into the unarchived source files using a patch or diff file that lists the changes. The doc files and/or README file will inform you should this be the case. The normal syntax for invoking Larry Wall's powerful *patch* utility is **patch < patchfile**.

You may now proceed to the build stage of the process.

3.Using Make

The Makefile is the key to the build process. In its simplest form, a Makefile is a script for compiling or building the "binaries", the executable portions of a package. The Makefile can also provide a means of updating a software package without having to recompile every single source file in it, but that is a different story (or a different article).

At some point, the Makefile launches cc or gcc. This is actually a preprocessor, a C (or C++) compiler, and a linker, invoked in that order. This process converts the source into the binaries, the actual executables.

Invoking *make* usually involves just typing **make**. This generally builds all the necessary executable files for the package in question. However, make can also do other tasks, such as installing the files in their proper directories (**make install**) and removing stale object files (**make clean**). Running **make −n** permits previewing the build process, as it prints out all the commands that would be triggered by a make, without actually executing them.

Only the simplest software uses a generic Makefile. More complex installations require tailoring the Makefile according to the location of libraries, include files, and resources on your particular machine. This is especially the case when the build needs the X11 libraries to install. *Imake* and *xmkmf* accomplish this task.

An `Imakefile` is, to quote the man page, a "template" Makefile. The imake utility constructs a Makefile appropriate for your system from the Imakefile. In almost all cases, however, you would run **xmkmf**, a shell script that invokes imake, a front end for it. Check the README or INSTALL file included in the software archive for specific instructions. (If, after dearchiving the source files, there is an `Imake` file present in the base directory, this is a dead giveaway that **xmkmf** should be run.) Read the `Imake` and `xmkmf` man pages for a more detailed analysis of the procedure.

Be aware that `xmkmf` and `make` may need to be invoked as root, especially when doing a **make install** to move the binaries over to the `/usr/bin` or `/usr/local/bin` directories. Using make as an ordinary user without root privileges will likely result in *write access denied* error messages because you lack write permission to system directories. Check also that the binaries created have the proper execute permissions for you and any other appropriate users.

Invoking **xmkmf** uses the `Imake` file to build a new Makefile appropriate for your system. You would normally invoke **xmkmf** with the **−a** argument, to automatically do a *make Makefiles, make includes,* and *make depend*. This sets the variables and defines the library locations for the compiler and linker. Sometimes, there will be no `Imake` file, instead there will be an `INSTALL` or `configure` script that will accomplish this purpose. Note that if you run `configure`, it should be invoked as **./configure** to ensure that the correct `configure` script in the current directory is called. In most cases, the `README` file included with the distribution will explain the install procedure.

It is usually a good idea to visually inspect the `Makefile` that `xmkmf` or one of the install scripts builds. The Makefile will normally be correct for your system, but you may occasionally be required to "tweak" it or correct errors manually.

Installing the freshly built binaries into the appropriate system directories is usually a matter of running **make install** as root. The usual directories for system−wide binaries on modern Linux distributions are `/usr/bin`, `/usr/X11R6/bin`, and `/usr/local/bin`. The preferred directory for new packages is `/usr/local/bin`, as this will keep separate binaries not part of the original Linux installation.

Packages originally targeted for commercial versions of UNIX may attempt to install in the `/opt` or other unfamiliar directory. This will, of course, result in an installation error if the intended installation directory does not exist. The simplest way to deal with this is to create, as root, an `/opt` directory, let the package install there, then add that directory to the `PATH` environmental variable. Alternatively, you may create symbolic links to the `/usr/local/bin` directory.

Your general installation procedure will therefore be:

- Read the `README` file and other applicable docs.
- Run **xmkmf −a**, or the `INSTALL` or `configure` script.
- Check the `Makefile`.
- If necessary, run **make clean**, **make Makefiles**, **make includes**, and **make depend**.
- Run **make**.
- Check file permissions.
- If necessary, run **make install**.

Notes:

* You would not normally build a package as root. Doing an **su** to root is only necessary for installing the compiled binaries into system directories.
* After becoming familiar with *make* and its uses, you may wish to add additional optimization options passed to `gcc` in the standard `Makefile` included or created in the package you are installing. Some of these common options are *−O2*, *−fomit−frame−pointer*, *−funroll−loops*, and *−mpentium* (if you are running a Pentium cpu). Use caution and good sense when modifying a `Makefile`!
* After the *make* creates the binaries, you may wish to **strip** them. The **strip** command removes the symbolic debugging information from the binaries, and reduces their size, often drastically. This also disables debugging, of course.
* The Pack Distribution Project offers a different approach to creating archived software packages, based on a set of Python scripting tools for managing symbolic links to files installed in separate *collection directories*. These archives are ordinary *tarballs*, but they install in `/coll` and `/pack` directories. You may find it necessary to download the *Pack−Collection* from the above site should you ever run across one of these distributions.

4.Prepackaged Binaries

4.1 Whats wrong with rpms?

Manually building and installing packages from source is apparently so daunting a task for some Linux users that they have embraced the popular *rpm* and *deb* or the newer Stampede *slp* package formats. While it may be the case that an *rpm* install normally runs as smoothly and as fast as a software install in a certain other notorious operating system, some thought should certainly be given to the disadvantages of self−installing, prepackaged binaries.

First, be aware that software packages are normally released first as "tarballs", and that prepackaged binaries follow days, weeks, even months later. A current *rpm* package is typically at least a couple of minor version behind the latest "tarball". So, if you wish to keep up with all the 'bleeding edge' software, you might not wish to wait for an *rpm* or *deb* to appear. Some less popular packages may never be *rpm*'ed.

Second, the "tarball" package may well be more complete, have more options, and lend itself better to customization and tweaking. The binary rpm version may be missing some of the functionality of the full release. Source *rpm*'s contain the full source code and are equivalent to the corresponding "tarballs", and they likewise need to be built and installed using either of the **rpm −−recompile packagename.rpm** or **rpm −−rebuild packagename.rpm** options.

Third, some prepackaged binaries will not properly install, and even if they do install, they could crash and core−dump. They may depend on different library versions than are present in your system, or they may be improperly prepared or just plain broken. In any case, when installing an *rpm* or *deb* you necessarily trust the expertise of the persons who have packaged it.

Finally, it helps to have the source code on hand, to be able to tinker with and learn from it. It is much more straightforward to have the source in the archive you are building the binaries from, and not in a separate source *rpm*.

Installing an *rpm* package is not necessarily a no–brainer. If there is a dependency conflict, an *rpm* install will fail. Likewise, should the *rpm* require a different version of libraries than the ones present on your system, the install may not work, even if you create symbolic links to the missing libraries from the ones in place. Despite their convenience, *rpm* installs often fail for the same reasons "tarball" ones do.

You must install *rpm*'s and *deb*'s as root, in order to have the necessary write permissions, and this opens a potentially serious security hole, as you may inadvertently clobber system binaries and libraries, or even install a *Trojan horse* that might wreak havoc upon your system. It is therefore important to obtain *rpm* and *deb* packages from a "trusted source". In any case, you should run a 'signature check' (against the MD5 checksum) on the package, **rpm −−checksig packagename.rpm**, before installing. Likewise highly recommended is running **rpm −K −−nopgp packagename.rpm**. The corresponding commands for *deb* packages are **dpkg −I | −−info packagename.deb** and **dpkg −e | −−control packagename.deb**.

- `rpm --checksig gnucash-1.1.23-4.i386.rpm`

  `gnucash-1.1.23-4.i386.rpm: size md5 OK`
- `rpm -K --nopgp gnucash-1.1.23-4.i386.rpm`

  `gnucash-1.1.23-4.i386.rpm: size md5 OK`

For the truly paranoid (and, in this case there is much to be said for paranoia), there are the *unrpm* and *rpmunpack* utilities available from the Sunsite utils/package directory for unpacking and checking the individual components of the packages.

Klee Diene has written an experimental *dpkgcert* package for verifying the integrity of installed *.deb* files against MD5 checksums. It is available from the Debian ftp archive. The current package name / version is *dpkgcert_0.2−4.1_all.deb*. The Jim Pick Software site maintains an experimental server database to provide *dpkgcert* certificates for the packages in a typical Debian installation.

In their most simple form, the commands **rpm −i packagename.rpm** and **dpkg −−install packagename.deb** automatically unpack and install the software. Exercise caution, though, since using these commands blindly may be dangerous to your system's health!

Note that the above warnings also apply, though to a lesser extent, to Slackware's *pkgtool* installation utility. All "automatic" software installations require caution.

The martian and alien programs allow conversion between the *rpm*, *deb*, Stampede *slp*, and *tar.gz* package formats. This makes these packages accessible to all Linux distributions.

Carefully read the man pages for the *rpm* and *dpkg* commands, and refer to the RPM HOWTO, TFUG's Quick Guide to Red Hat's Package Manager, and The Debian Package Management Tools for more detailed information.

4.2 Problems with rpms: an example

Jan Hubicka wrote a very nice fractal package called *xaos*. At his home page, both `.tar.gz` and `rpm` packages are available. For the sake of convenience, let us try the rpm version, rather than the "tarball".

Unfortunately, the rpm of *xaos* fails to install. Two separate rpm versions misbehave.

rpm −i −−test XaoS−3.0−1.i386.rpm

```
error: failed dependencies:
        libslang.so.0 is needed by XaoS-3.0-1
        libpng.so.0 is needed by XaoS-3.0-1
        libaa.so.1 is needed by XaoS-3.0-1
```

rpm −i −−test xaos−3.0−8.i386.rpm

```
error: failed dependencies:
        libaa.so.1 is needed by xaos-3.0-8
```

The strange thing is that `libslang.so.0`, `libpng.so.0`, and `libaa.so.1` are all present in `/usr/lib` on the system tested. The rpms of *xaos* must have been built with slightly different versions of those libraries, even if the release numbers are identical.

As a test, let us try installing `xaos-3.0-8.i386.rpm` with the −−*nodeps* option to force the install. A trial run of *xaos* crashes.

```
xaos: error in loading shared libraries: xaos: undefined symbol: __fabsl
```

Let us stubbornly try to get to the bottom of this. Running *ldd* on the *xaos* binary to find its library dependencies shows all the necessary shared libraries present. Running *nm* on the `/usr/lib/libaa.so.1` library to list its symbolic references shows that it is indeed missing *__fabsl*. Of course, the absent reference *could* be missing from one of the other libraries... There is nothing to be done about that, short of replacing one or more libraries.

Enough! Download the "tarball", `XaoS-3.0.tar.gz`, available from the ftp site, as well as from the home page. Try building it. Running **./configure**, **make**, and finally (as root) **make install**, works flawlessly.

This is one of an number of examples of prepackaged binaries being more trouble than they are worth.

5.Termcap and Terminfo Issues

According to its man page, *"terminfo is a data base describing terminals, used by screen−oriented programs..."*. It defines a generic set of control sequences (escape codes) used to display text on terminals, and makes possible support for different terminal hardware without the need for special drivers. The *terminfo* libraries are located in `/usr/share/terminfo` on modern Linux distributions.

The *terminfo* database has largely supplanted the older *termcap* and the totally obsolete *termlib* ones. This is usually of no concern for program installation except when dealing with a package that requires *termcap*.

Most Linux distributions now use *terminfo*, but still retain the older termcap libraries for compatibility with legacy applications (see `/etc/termcap`). Sometimes there is a special compatibility package that needs to be installed to facilitate use of termcap linked binaries. Very occasionally, an *#define termcap* statement might need to be commented out of a source file. Check the appropriate doc files for your particular distribution for definitive information on this.

6.Backward Compatibility With a.out Binaries

In a very few cases, it is necessary to use a.out binaries, either because the source code is not available or because it is not possible to build new ELF binaries from the source for some reason.

As it happens, ELF installations almost always have a complete set of a.out libraries in the `/usr/i486−linuxaout/lib` directory. The numbering scheme for a.out libraries differs from that of ELF ones, cleverly avoiding conflicts that could cause confusion. The a.out binaries should therefore be able to find the correct libraries at runtime, but this might not always be the case.

Note that the kernel needs to have a.out support built into it, either directly or as a loadable module. It may be necessary to rebuild the kernel to enable this. Moreover, some Linux distributions require installation of a special compatibility package, such as Debian's `xcompat` for executing a.out X applications.

6.1 An Example

Jerry Smith wrote a very handy *rolodex* program some years back. It uses the Motif libraries, but fortunately is available as a statically linked binary in a.out format. Unfortunately, the source requires numerous tweaks to rebuild using the *lesstif* libraries. Even more unfortunately, the a.out binary bombs on an ELF system with the following error message.

```
xrolodex: can't load library '//lib/libX11.so.3'
No such library
```

As it happens, there is such a library, in `/usr/i486-linuxaout/lib`, but xrolodex is unable to locate it at run time. The simple solution is to provide a symbolic link in the `/lib` directory:

```
ln -s /usr/i486-linuxaout/lib/X11.so.3.1.0 libX11.so.3
```

It turns out to be necessary to provide similar links for the libXt.so.3 and libc.so.4 libraries. This needs to be done as root, of course. Note that you should make absolutely certain you will not overwrite or cause version number conflicts with pre-existing libraries. Fortunately, the new ELF libraries have higher version numbers than the older a.out ones, to anticipate and forestall just such problems.

After creating the three links, *xrolodex* runs fine.

The *xrolodex* package was originally posted on Spectro, but seems to vanished from there. It may currently be downloaded from Sunsite as a *tar.Z* format source file [512k].

7.Troubleshooting

If *xmkmf* and/or *make* succeeded without errors, you may proceed to the next section. However, in "real life", few things work right the first time. This is when your resourcefulness is put to the test.

7.1 Link Errors

- Suppose *make* fails with a `Link error: -lX11: No such file or directory`, even after xmkmf has been invoked. This may mean that the *Imake* file was not set up properly. Check the first part of the *Makefile* for lines such as:

  ```
  LIB=            -L/usr/X11/lib
  INCLUDE=        -I/usr/X11/include/X11
  LIBS=           -lX11 -lc -lm
  ```

  The `-L` and `-I` switches tell the compiler and linker where to look for the *library* and *include* files, respectively. In this example, the *X11 libraries* should be in the `/usr/X11/lib` directory, and the *X11 include files* should be in the `/usr/X11/include/X11` directory. If this is incorrect for your machine, make the necessary changes to the *Makefile* and try the *make* again.

* Undefined references to math library functions, such as the following:

```
/tmp/cca011551.o(.text+0x11): undefined reference to `cos'
```

  The fix for this is to explicitly link in the `math library`, by adding an **–lm** to the *LIB* or *LIBS* flags in the `Makefile` (see previous example).

* Yet another thing to try if *xmkmf* fails is the following script:

```
make –DUseInstalled –I/usr/X386/lib/X11/config
```

  This is a sort of bare bones equivalent of *xmkmf*.

* In a very few cases, running *ldconfig* as *root* may be the solution:

  # ldconfig updates the shared library symbolic links. *This may not be necessary* .

* Some `Makefiles` use unrecognized aliases for libraries present in your system. For example, the build may require `libX11.so.6`, but there exists no such file or link in `/usr/X11R6/lib`. Yet, there is a `libX11.so.6.1`. The solution is to do a **ln –s /usr/X11R6/lib/libX11.so.6.1 /usr/X11R6/lib/libX11.so.6**, as root. This may need to be followed by a **ldconfig**.

* Sometimes the source needs the older release X11R5 libraries to build. If you have the R5 libs in /usr/X11R6/lib (you were given the option of having them when first installing Linux), then you need only ensure that you have the links that the software needs to build. The `R5 libs` are named `libX11.so.3.1.0`, `libXaw.so.3.1.0`, and `libXt.so.3.1.0`. You generally need links, such as *libX11.so.3 –> libX11.so.3.1.0*. Possibly the software will also need a link of the form *libX11.so –> libX11.so.3.1.0*. Of course, to create a "missing" link, use the command **ln –s libX11.so.3.1.0 libX11.so**, *as root*.

* Some packages will require you to install updated versions of one or more libraries. For example, the 4.x versions of the *StarOffice* suite from StarDivision GmbH were notorious for needing a `libc` version 5.4.4 or greater. Even the more recent *StarOffice* 5.0 will not run after installation with the new `glibc 2.1` libs. Fortunately, the newer *StarOffice* 5.1 solves these problems. If running an older version of *StarOffice* you would, as *root*, need to copy one or more libraries to the appropriate directories, remove the old libraries, then reset the symbolic links (check the latest version of the `StarOffice miniHOWTO` for more information on this). **Caution: Exercise extreme care in**

this, as you can render your system nonfunctional if you screw up. You can usually find the latest updated libraries at Sunsite.

7.2 Other Problems

- An installed *Perl* or shell script gives you a `No such file or directory` error message. In this case, check the file permissions to make sure the file is executable and check the file header to ascertain whether the shell or program invoked by the script is in the place specified. For example, the scrip may begin with:

  ```
  #!/usr/local/bin/perl
  ```

  If *Perl* is in fact installed in your `/usr/bin` directory instead of the `/usr/local/bin` one, then the script will not run. There are two methods of correcting this. The script file header may be changed to `#!/usr/bin/perl`, or a symbolic link to the correct directory may be added, **ln −s /usr/bin/perl /usr/local/bin/perl**.

- Some X11 software requires the Motif libraries to build. The standard Linux distributions do not have the Motif libraries installed, and at present Motif costs an extra $100−$200 (though the freeware Lesstif also works in many cases). If you need Motif to build a certain package, but lack the Motif libraries, it may be possible to obtain *statically linked binaries*. Static linking incorporates the library routines in the binaries themselves. This results in much larger binary files, but the code will run on systems lacking the libraries.

  When a package requires libraries not present on your system for the build, it will result in link errors (`undefined reference` errors). The libraries may be expensive proprietary ones or difficult to find for sone other reason. In that case, obtaining a *statically linked* binary either from the author of the package or from a Linux user group may be the easiest to implement fix.

- Running a *configure* script creates a strange Makefile, one seemingly unrelated to the package you are attempting to build. This means the wrong *configure* ran, one found somewhere else in your path. Always invoke *configure* as **./configure** to prevent this.

- Most Linux distributions have changed over to the `libc 6 / glibc 2` libraries from the older `libc 5`. Precompiled binaries that worked with the older library may bomb if you have upgraded your library. The solution is to either recompile the applications from the source or to obtain newer precompiled binaries. If you are in the process of upgrading your system to `libc 6` and are experiencing problems, refer to Eric Green's *Glibc 2 HOWTO*.

Note that there are some minor incompatibilities between `glibc` versions, so a binary built with `glibc 2.1` may not work with `glibc 2.0`, and vice versa.

- Sometimes it is necessary to remove the *−ansi* option from the compile flags in the `Makefile`. This enables gcc's extra, non−ANSI features, and allows building packages that require these extensions. (Thanks to Sebastien Blondeel for pointing this out.)

- Some programs require having *setuid root*, in order to run with *root privileges*. The command to implement this is **chmod u+s filename**, *as root* (note that the program must already be owned by root). This has the effect of setting the *setuid* bit in the file permissions. This issue comes up when the program accesses the system hardware, such as a modem or CD ROM drive, or when the SVGA libs are invoked from console mode, as in one particularly notorious emulation package. If a program works when run by root, but gives *access denied* error messages to an ordinary user, suspect this as the cause.

  Warning: A program with *setuid* as root may pose a security risk to your system. The program runs with root privileges and thus has the potential for doing significant damage. Make certain that you know what the program does, by looking at the source if possible, before setting the *setuid* bit.

7.3 Tweaking and fine tuning

You may wish to examine the `Makefile` to make certain that the best compilation options for your system are invoked. For example, setting the *−O2* flag chooses the highest level of optimization and the *−fomit−frame−pointer* flag results in a smaller binary (though debugging will then be disabled). **Do not play around with this unless you know what you are doing, and in any case, not until after a trial *build* works.**

7.4 Where to go for more help

In my experience, perhaps 25% of applications build "right out of the box". Another 50% or so can be "persuaded" to build with an effort ranging from trivial to herculean. That still means a significant number of packages will not build no matter what. Even then, the Intel `ELF` and/or `a.out` binaries for these might possibly be found at Sunsite or the TSX−11 archive. Red Hat and Debian have extensive archives of prepackaged binaries of most of the popular Linux software. Perhaps the author of the software can supply the binaries compiled for your particular flavor of machine.

`Note that if you obtain precompiled binaries, you will need to check for` compatibility with your system:

- The binaries must run on your hardware (i.e., Intel x86).
- The binaries must be compatible with your kernel (i.e., a.out or ELF).
- Your libraries must be up to date.
- Your system must have the appropriate installation utility (rpm or deb).

If all else fails, you may find help in the appropriate newsgroups, such as comp.os.linux.x or comp.os.linux.development.

If nothing at all works, at least you gave it your best effort, and you learned a lot.

8.Final Steps

Read the software package documentation to determine whether certain environmental variables need setting (in `.bashrc` or `.cshrc`) and if the `.Xdefaults` and `.Xresources` files need customizing.

There may be an applications default file, usually named `Xfoo.ad` in the original Xfoo distribution. If so, edit the Xfoo.ad file to customize it for your machine, then rename (**mv**) it Xfoo and install it in the `/usr/lib/X11/app-defaults` directory, *as root*. Failure to do this may cause the software to behave strangely or even refuse to run.

Most software packages come with one or more preformatted man pages. *As root*, copy the Xfoo.man file to the appropriate `/usr/man`, `/usr/local/man`, or `/usr/X11R6/man` directory (`man1` – `man9`), and rename it accordingly. For example, if Xfoo.man ends up in /usr/man/man4, it should be renamed Xfoo.4 (mv Xfoo.man Xfoo.4). By convention, user commands go in `man1`, games in `man6`, and administration packages in `man8` (see the *man docs* for more details). Of course, you may deviate from this on your own system, if you like.

A few packages will not install the binaries in the appropriate system directories, that is, they are missing the *install* option in the `Makefile`. Should this be the case, you can install the binaries manually by copying the binaries to the appropriate system directory, `/usr/bin`, `/usr/local/bin` or `/usr/X11R6/bin`, *as root*, of course. Note that `/usr/local/bin` is the preferred directory for binaries that are not part of the Linux distribution's base install.

Some or all of the above procedures should, in most cases, be handled automatically by a **make install**, and possibly a **make install.man** or **make install_man**. If so, the `README` or `INSTALL` doc file will specify this.

9.First Example: Xscrabble

Matt Chapman's `Xscrabble` seemed like a program that would be interesting to have, since I happen to be an avid Scrabble`TM` player. I downloaded it, uncompressed it, and built it following the procedure in the README file:

```
xmkmf
make Makefiles
make includes
make
```

Of course it did not work...

```
gcc -o xscrab -O2 -O -L/usr/X11R6/lib
init.o xinit.o misc.o moves.o cmove.o main.o xutils.o mess.o popup.o
widgets.o display.o user.o CircPerc.o
-lXaw -lXmu -lXExExt -lXext -lX11 -lXt -lSM -lICE -lXExExt -lXext -lX11
-lXpm -L../Xc -lXc

BarGraf.o(.text+0xe7): undefined reference to `XtAddConverter'
BarGraf.o(.text+0x29a): undefined reference to `XSetClipMask'
BarGraf.o(.text+0x2ff): undefined reference to `XSetClipRectangles'
BarGraf.o(.text+0x375): undefined reference to `XDrawString'
BarGraf.o(.text+0x3e7): undefined reference to `XDrawLine'
etc.
etc.
etc...
```

I enquired about this in the comp.os.linux.x newsgroup, and someone kindly pointed out that apparently the Xt, Xaw, Xmu, and X11 libs were not being found at the link stage. Hmmm...

There were two main Makefiles, and the one in the `src` directory caught my interest. One line in the Makefile defined LOCAL_LIBS as: LOCAL_LIBS = $(XAWLIB) $(XMULIB) $(XTOOLLIB) $(XLIB) Here were references to the libs not being found by the linker.

Looking for the next reference to LOCAL_LIBS, I saw on line 495 of that Makefile:

```
      $(CCLINK) -o $@ $(LDOPTIONS) $(OBJS) $(LOCAL_LIBS) $(LDLIBS)
$(EXTRA_LOAD_FLAGS)
```

Now what were these LDLIBS?

```
      LDLIBS = $(LDPOSTLIB) $(THREADS_LIBS) $(SYS_LIBRARIES)
$(EXTRA_LIBRARIES)
```

The SYS_LIBRARIES were:

```
SYS_LIBRARIES = -lXpm -L../Xc -lXc
```

Yes! Here were the missing libraries.

Possibly the linker needed to see the LDLIBS before the LOCAL_LIBS... So, the first thing to try was to modify the Makefile by transposing the $(LOCAL_LIBS) and $(LDLIBS) on line 495, so it would now read:

```
        $(CCLINK) -o $@ $(LDOPTIONS) $(OBJS) $(LDLIBS) $(LOCAL_LIBS)
$(EXTRA_LOAD_FLAGS)                      ^^^^^^^^^^^^^^^^^^^^^^^
```

I tried running *make* again with the above change, and lo and behold, it worked this time. Of course, Xscrabble still needed some fine tuning and twiddling, such as renaming the dictionary and commenting out some assert statements in one of the source files, but since then it has provided me with many hours of pleasure.

[Note that a newer version of Xscrabble is now available in rpm format, and this installs without problems.]

You may e−mail Matt Chapman, and download *Xscrabble* from his home page.

```
Scrabble is a registered trademark of the Milton Bradley Co., Inc.
```

10.Second Example: Xloadimage

This example poses an easier problem. The *xloadimage* program seemed a useful addition to my set of graphic tools. I copied the `xloadi41.gz` file directly from the source directory on the CD included with the excellent X User Tools book, by Mui and Quercia. As expected, *tar xzvf* unarchives the files. The *make*, however, produces a nasty−looking error and terminates.

```
gcc -c -O -fstrength-reduce -finline-functions -fforce-mem
-fforce-addr -DSYSV  -I/usr/X11R6/include
-DSYSPATHFILE=\"/usr/lib/X11/Xloadimage\" mcidas.c

In file included from /usr/include/stdlib.h:32,
                 from image.h:23,
                 from xloadimage.h:15,
                 from mcidas.c:7:
/usr/lib/gcc-lib/i486-linux/2.6.3/include/stddef.h:215:
conflicting types for `wchar_t'
/usr/X11R6/include/X11/Xlib.h:74: previous declaration of
`wchar_t'
make[1]: *** [mcidas.o] Error 1
make[1]: Leaving directory
`/home/thegrendel/tst/xloadimage.4.1'
make: *** [default] Error 2
```

The error message contains the essential clue.

Looking at the file `image.h`, line 23...

```
#include <stdlib.h>
```

Aha, somewhere in the source for *xloadimage*, *wchar_t* has been redefined from what was specified in the standard include file, `stdlib.h`. Let us first try commenting out line 23 in `image.h`, as perhaps the *stdlib.h include* is not, after all, necessary.

At this point, the build proceeds without any fatal errors. The *xloadimage* package functions correctly now.

11.Third Example: Fortune

This example requires some knowledge of C programming. The majority of UNIX/Linux software is written in C, and learning at least a little bit of C would certainly be an asset for anyone serious about software installation.

The notorious *fortune* program displays up a humorous saying, a "fortune cookie", every time Linux boots up. Unfortunately (pun intended), attempting to build fortune on a Red Hat distribution with a 2.0.30 kernel generates fatal errors.

```
~/fortune# make all


gcc -O2 -Wall -fomit-frame-pointer -pipe   -c fortune.c -o
```

```
fortune.o
fortune.c: In function `add_dir':
fortune.c:551: structure has no member named `d_namlen'
fortune.c:553: structure has no member named `d_namlen'
make[1]: *** [fortune.o] Error 1
make[1]: Leaving directory `/home/thegrendel/for/fortune/fortune'
make: *** [fortune-bin] Error 2
```

Looking at `fortune.c`, the pertinent lines are these.

```
   if (dirent->d_namlen == 0)
            continue;
        name = copy(dirent->d_name, dirent->d_namlen);
```

We need to find the structure `dirent`, but it is not declared in the *fortune.c* file, nor does a **grep dirent** show it in any of the other source files. However, at the top of *fortune.c*, there is the following line.

```
#include <dirent.h>
```

This appears to be a system library include file, therefore, the logical place to look for *dirent.h* is in */usr/include*. Indeed, there does exist a *dirent.h* file in */usr/include*, but that file does not contain the declaration of the `dirent` structure. There is, however, a reference to another *dirent.h* file.

```
#include <linux/dirent.h>
```

At last, going to */usr/include/linux/dirent.h*, we find the structure declaration we need.

```
struct dirent {
        long            d_ino;
        __kernel_off_t  d_off;
        unsigned short  d_reclen;
        char            d_name[256]; /* We must not include
limits.h! */
};
```

Sure enough, the structure declaration contains no *d_namelen*, but there are a couple of "candidates" for its equivalent. The most likely of these is *d_reclen*, since this structure member probably represents the length of something and it is a short integer. The other possibility, *d_ino*, could be an inode number, judging by its name and type. As a matter of fact, we are probably dealing with a "directory entry" structure, and these elements represent attributes of a file, its name, inode, and length (in blocks). This would seem to validate our guess.

Let us edit the file `fortune.c`, and change the two *d_namelen* references in lines 551 and 553 to *d_reclen*. Try a *make all* again. **Success.** It builds without errors. We can now get our "cheap thrills" from fortune.

12.Fourth Example: Hearts

Here is the hoary old game of Hearts, written for UNIX systems by Bob Ankeney sometime in the '80's, revised in 1992 by Mike Yang, and currently maintained by Jonathan Badger. Its predecessor was an even older Pascal program by Don Backus of Oregon Software, later updated by Jeff Hemmerling. Originally intended as a multiplayer client, it also works well in single–player mode against computer opponents. The graphics are nice, though the game lacks sophisticated features and the computer players are not particularly strong. All the same, it seems to be the only decent Hearts game available for UNIX and Linux machines even at this late date.

Due to its age and lineage, this package is particularly difficult to build on a Linux system. It requires solving a long and perplexing series of puzzles. It is an exercise in patience and determination.

Before beginning, make certain that you have either the `motif` or `lesstif` libraries installed.

- **xmkmf**

  make

```
client.c: In function `read_card':
client.c:430: `_tty' undeclared (first use in this function)
client.c:430: (Each undeclared identifier is reported only once
client.c:430: for each function it appears in.)
client.c: In function `scan':
client.c:685: `_tty' undeclared (first use in this function)
make: *** [client.o] Error 1
```

These are the culprits in the file `client.c`:

```
#ifndef SYSV
        (buf[2] != _tty.sg_erase) && (buf[2] != _tty.sg_kill)) {
 #else
        (buf[2] != CERASE) && (buf[2] != CKILL)) {
```

```
#endif
```

- In `client.c`, add

```
#define SYSV
```

at line 39. This will bypass the reference to *_tty*.

make

```
client.c:41: sys/termio.h: No such file or directory
make: *** [client.o] Error 1
```

- The include file `termio.h` is in the `/usr/include` directory on a Linux system, rather than the `/usr/include/sys` one, as was the case on older UNIX machines. Therefore, change line 41 of `client.c` from

```
#include <sys/termio.h>
```

to

```
#include <termio.h>
```

make

```
gcc -o hearts -g      -L/usr/X11R6/lib client.o hearts.o select.o connect.o
sockio.o start_dist.o  -lcurses -ltermlib
/usr/bin/ld: cannot open -ltermlib: No such file or directory
collect2: ld returned 1 exit status
make: *** [hearts] Error 1
```

-

Modern Linux distributions use the *terminfo* and/or *termcap* database, rather than the obsolete *termlib* one.

Edit the `Makefile`.

Line 655:

```
CURSES_LIBRARIES = -lcurses -ltermlib
```

changes to:

```
CURSES_LIBRARIES = -lcurses -ltermcap
```

make

```
gcc -o xmhearts -g      -L/usr/X11R6/lib xmclient.o hearts.o select.o
connect.o sockio.o start_dist.o gfx.o  -lXm_s -lXt -lSM -lICE -lXext -lX11
-lPW
/usr/bin/ld: cannot open -lXm_s: No such file or directory
collect2: ld returned 1 exit status
```

-

The main *lesstif* library is `libXm`, rather than `libXm_s`. Therefore, edit the `Makefile`.

In line 653:

```
XMLIB = -lXm_s $(XTOOLLIB) $(XLIB) -lPW
```

changes to:

```
XMLIB = -lXm $(XTOOLLIB) $(XLIB) -lPW
```

make

```
gcc -o xmhearts -g      -L/usr/X11R6/lib xmclient.o hearts.o select.o
connect.o sockio.o start_dist.o gfx.o  -lXm -lXt -lSM -lICE -lXext -lX11 -lPW
/usr/bin/ld: cannot open -lPW: No such file or directory
```

```
collect2: ld returned 1 exit status
make: *** [xmhearts] Error 1
```

- Round up the usual suspects.

There is no `PW` library. Edit the `Makefile`.

Line 653:

```
XMLIB = -lXm $(XTOOLLIB) $(XLIB) -lPW
```

changes to:

```
XMLIB = -lXm $(XTOOLLIB) $(XLIB) -lPEX5
```

(The `PEX5` lib comes closest to `PW`.)

make

```
rm -f xmhearts
gcc -o xmhearts -g      -L/usr/X11R6/lib xmclient.o hearts.o select.o
connect.o sockio.o start_dist.o gfx.o  -lXm -lXt -lSM -lICE -lXext -lX11 -lPEX5
```

The `make` finally works (hurray!).

- *Installation:*

As root,

```
[root@localhost hearts]# make install
install -c -s  hearts /usr/X11R6/bin/hearts
install -c -s  xmhearts /usr/X11R6/bin/xmhearts
install -c -s  xawhearts /usr/X11R6/bin/xawhearts
install in . done
```

Test run:

rehash

(We're running the `tcsh` shell.)

xmhearts

```
localhost:~/% xmhearts
Can't invoke distributor!
```

- From README file in the `hearts` package:

```
Put heartsd, hearts_dist, and hearts.instr in the HEARTSLIB
directory defined in local.h and make them world-accessible.
```

From the file `local.h`:

```
/* where the distributor, dealer and instructions live */

#define HEARTSLIB "/usr/local/lib/hearts"
```

This is a classic case of RTFM.

As *root*,

cd /usr/local/lib

mkdir hearts

cd !$

Copy the `distributor` files to this directory.

cp /home/username/hearts/heartsd .

cp /home/username/hearts/hearts_dist .

cp /home/username/hearts/hearts.instr .

- *Try another test run.*

xmhearts

It works some of the time, but more often than not crashes with a `dealer died!` message.

- The "distributor" and "dealer" scan the hardware ports. We should thus suspect that those programs need root user privileges.

Try, as *root*,

chmod u+s /usr/local/lib/heartsd

chmod u+s /usr/local/lib/hearts_dist

(Note that, as previously discussed, *suid* binaries may create security holes.)

xmhearts

It finally works!

Hearts is available from Sunsite.

13.Fifth Example: XmDipmon

```
Bullwinkle: Hey Rocky, watch me pull a rabbit out of my hat.
Rocky:      But that trick never works.
Bullwinkle: This time for sure.
            Presto!
            Well, I'm gettin' close.
Rocky:      And now it's time for another special feature.
            --- "Rocky and His Friends"
```

XmDipmon is a nifty little application that displays a button showing the status of an Internet connection. It flashes and beeps when the connection is broken, as is all too often the case in on rural telephone systems. Unfortunately, XmDipmon works only with *dip*, making it useless for those people, the majority, who use *chat* to connect.

Building XmDipmon is not a problem. XmDipmon links to the *Motif* libraries, but it builds and works fine with *Lesstif*. The challenge is to alter the package to work when using *chat*. This involves actually tinkering with the source code, and necessarily requires some programming knowledge.

```
"When xmdipmon starts up, it checks for a file called /etc/dip.pid
 (you can let it look at another file by using the -pidfile
 command line option).  This file contains the PID of the dip
 deamon (dip switches itself into deamon mode once it has
 established a connection)."
              --- from the XmDipmon README file
```

Using the *−pidfile* option, the program can be directed to check for a different file upon startup, one that exists only during a successful *chat* login. The obvious candidate is the modem lock file. We could therefore try invoking the program with **xmdipmon −pidfile /var/lock/LCK..ttyS3** (this assumes that the modem is on com port #4, ttyS3). This only solves part of the problem, however. The program continually monitors the *dip daemon*, and we need to change this so it instead polls a process associated with *chat* or *ppp*.

There is only a single source file, and fortunately it is well−commented. Scanning the `xmdipmon.c` file, we find the *getProcFile* function, whose header description reads as follows.

```
/*****
* Name:                 getProcFile
* Return Type:  Boolean
* Description:  tries to open the /proc entry as read from the dip pid file.
<snip>
*****/
```

We are hot on the trail now. Tracing into the body of the function...

```
/* we watch the status of the real dip deamon */
sprintf(buf, "/proc/%i/status", pid);
procfile = (String)XtMalloc(strlen(buf)*sizeof(char)+1);
strcpy(procfile, buf);
procfile[strlen(buf)] = '\0';
```

The culprit is line 2383:

```
                    sprintf(buf, "/proc/%i/status", pid);
                                  ^^^^^^^^^^^^^^^^^^^^
```

This checks whether the dip daemon process is running . So, how can we change this to monitor the pppd daemon instead?

Looking at the *pppd* manpage:

```
FILES
       /var/run/pppn.pid (BSD or Linux), /etc/ppp/pppn.pid (others)
                     Process-ID for pppd process on ppp interface unit n.
```

Change line 2383 in `xmdipmon.c` to:

```
                    sprintf(buf, "/var/run/ppp0.pid" );
```

Rebuild the revised package. No problems with the build. Now test it with the new command line argument. It works like a charm. The little blue button indicates when a `ppp` connection to the ISP has been established, and flashes and beeps when the connection is broken. Now we have a fully functional *chat* monitor.

XmDipmon can be downloaded from Ripley Linux Tools.

14.Where to Find Source Archives

Now that you are eager to use your newly acquired knowledge to add utilities and other goodies to your system, you may find them online at the Linux Applications and Utilities Page, or on one of the very reasonably priced CD ROM archives by Red Hat, InfoMagic, Linux Systems Labs, Cheap Bytes, and others.

A comprehensive repository of source code is the comp sources UNIX archive.

Much UNIX source code is posted on the alt.sources newsgroup. If you are looking for particular source code packages, you may post on the related alt.sources.wanted newsgroup. Another good place to check is the comp.os.linux.announce newsgroup. To get on the Unix sources mailing list, send a *subscribe* message there.

Archives for the alt.sources newsgroup are at the following ftp sites:

- ftp.sterling.com/usenet/alt.sources/
- wuarchive.wustl.edu/usenet/alt.sources/articles
- src.doc.ic.ac.uk/usenet/alt.sources/articles

15.Final Words

To sum up, persistence makes all the difference (and a high frustration threshold certainly helps). As in all endeavors, learning from mistakes is critically important. Each misstep, every failure contributes to the body of knowledge that will lead to mastery of **the art of building software**.

16.References and Further Reading


```
BORLAND C++ TOOLS AND UTILITIES GUIDE, Borland International, 1992,
pp. 9-42.
[One of the manuals distributed with Borland C++, ver. 3.1. Gives
a fairly good intro to make syntax and concepts, using Borland's
crippled implementation for DOS.]

DuBois, Paul: SOFTWARE PORTABILITY WITH IMAKE, O'Reilly and Associates,
1996, ISBN 1-56592-226-3.
[This is reputed to be the definitive imake reference, though I did not
have it available when writing this article.]

Frisch, Aeleen: ESSENTIAL SYSTEM ADMINISTRATION (2nd ed.), O'Reilly and
Associates, 1995, ISBN 1-56592-127-5.
[This otherwise excellent sys admin handbook has only sketchy coverage
of software building.]

Hekman, Jessica: LINUX IN A NUTSHELL, O'Reilly and Associates, 1997, ISBN
1-56592-167-4.
[Good all-around reference to Linux commands.]

Lehey, Greg: PORTING UNIX SOFTWARE, O'Reilly and Associates, 1995, ISBN
1-56592-126-7.

Mayer, Herbert G.: ADVANCED C PROGRAMMING ON THE IBM PC, Windcrest Books,
1989, ISBN 0-8306-9363-7.
[An idea-filled book for the intermediate to advanced C programmer.
Superb coverage of algorithms, quirks of the language, and even
amusements.  Unfortunately, out of print.]

Mui, Linda and Valerie Quercia: X USER TOOLS, O'Reilly and Associates,
1994, ISBN 1-56592-019-8, pp. 734-760.

Oram, Andrew and Steve Talbott: MANAGING PROJECTS WITH MAKE, O'Reilly
and Associates, 1991, ISBN 0-937175-90-0.

Peek, Jerry and Tim O'Reilly and Mike Loukides: UNIX POWER TOOLS,
O'Reilly and Associates / Random House, 1997, ISBN 1-56592-260-3.
[A wonderful source of ideas, and tons of utilities you may end up
building from the source code, using the methods discussed in
this article.]
```

```
Stallman, Richard M. and Roland McGrath: GNU MAKE, Free Software
Foundation, 1995, ISBN 1-882114-78-7.
[Required reading.]

Waite, Mitchell, Stephen Prata, and Donald Martin: C PRIMER PLUS, Waite Group
Press, ISBN 0-672-22090-3,.
[Probably the best of the introductions to C programming.  Extensive
coverage for a primer.  Newer editions now available.]

Welsh, Matt and Lar Kaufman: RUNNING LINUX, O'Reilly and Associates,
1996, ISBN 1-56592-151-8.
[Still the best overall Linux reference, though lacking in depth
in some areas.]


The man pages for dpkg, gcc, gzip, imake, ldconfig, ldd, make, nm, patch,
rpm, shar, strip, tar, termcap, terminfo, and xmkmf.


The BZIP2 HOWTO, by David Fetter.

The Glibc2 HOWTO, by Eric Green

The LINUX ELF HOWTO, by Daniel Barlow.

The RPM HOWTO, by Donnie Barnes.

The StarOffice miniHOWTO, by Matthew Borowski.
```


[These HOWTOs should be in the `/usr/doc/HOWTO` or `/usr/doc/HOWTO/mini` directory on your system. Updated versions are available in text, HTML, and SGML format from the LDP site, and usually from the respective authors' home sites.]

17.Credits

The author of this HOWTO would like to thank the following persons for their helpful suggestions, corrections, and encouragement.

- R. Brock Lynn
- Michael Jenner
- Fabrizio Stefani

Kudos also go to the fine people who have translated this HOWTO into Italian and Japanese.

And, of course, thanks, praise, benedictions and hosannahs to Greg Hankins and Tim Bynum of the Linux Documentation Project, which has made all this possible.
