Software Release Practice HOWTO

Eric Steven Raymond

Thyrsus Enterprises

<esr@thyrsus.com>

Revision History

Revision 4.0	2010-04-11	Revised by: esr
It's no longer necessary to provide RPMS or debs at project level. The packaging infrastructure has gotten good at that. New caveats about configuration and autotools. AsciiDOC is now a viable alternative fr documentation masters.		
Revision 3.9	2004-11-28	Revised by: esr
New material on good patching practice. Recommend Electric Fence and valgrind rather than proprietary tools.		
Revision 3.8	2003-02-17	Revised by: esr
URL fixups after site move.		
Revision 3.7	2002-09-25	Revised by: esr
Point at the DocBook Demystification HOWTO.		
Revision 3.6	2002-09-12	Revised by: esr
Incorporated material on portability by Keith Bostic.		
Revision 3.6	2002-08-14	Revised by: esr
Rewrote section on documentation practice, since XML-Docbook is mature now.		
Revision 3.5	2002-07-04	Revised by: esr
Added section on providing checksums. Cited doclifter.		
Revision 3.4	2002-01-04	Revised by: esr
More about good patching practice.		
Revision 3.3	2001-08-16	Revised by: esr
New section about how to send good patches.		
Revision 3.2	2001-07-11	Revised by: esr
Note about not relying on proprietary components.		
Revision 3.1	2001-02-22	Revised by: esr
LDP Styleguide fixes.		
Revision 3.0	2000-08-12	Revised by: esr
First DocBook version. Advice on SourceForge and a major section on documentation practice added.		

This HOWTO describes good release practices for Linux and other open-source projects. By following these practices, you will make it as easy as possible for users to build your code and use it, and for other developers to understand your code and cooperate with you to improve it.

This document is a must-read for novice developers. Experienced developers should review it when they are about to release a new project. It will be revised periodically to reflect the evolution of good-practice standards.

Table of Contents

Table of Contents

1. Introduction

1.1. Why this document?

There is a large body of good-practice traditions for open-source code that helps other people port, use, and cooperate with developing it. Some of these conventions are traditional in the Unix world and predate Linux; others have developed recently in response to particular new tools and technologies such as the World Wide Web.

This document will help you learn good practice. It is organized into topic sections, each containing a series of checklist items. Think of these as a pre-flight checklist for your patch or software distribution.

A translation into German is available.

1.2. New versions of this document

This document will be posted monthly to the newsgroups comp.os.linux.answers. You can also view the latest version of this HOWTO on the World Wide Web via the URL http://www.en.tldp.org/HOWTO/Software-Release-Practice.html.

Feel free to mail any questions or comments about this HOWTO to Eric S. Raymond, `<esr@snark.thyrsus.com>`.

---

2. Good patching practice

Most people get involved in open-source software by writing patches for other peoples' software before releasing projects of their own. Suppose you've written a set of source-code changes for someone else's baseline code. Now put yourself in that person's shoes. How is he to judge whether to include the patch?

The truth is that it is very difficult to judge the quality of code. So developers tend to evaluate patches by the quality of the submission. They look for clues in the submitter's style and communications behavior instead — indications that the person has been in their shoes and understands what it's like to have to evaluate and merge an incoming patch.

This is actually a pretty reliable proxy for code quality. In many years of dealing with patches from many hundreds of strangers, I have only seldom seen a patch that was thoughtfully presented and respectful of my time but technically bogus. On the other hand, experience teaches me that patches which look careless or are packaged in a lazy and inconsiderate way are very likely to actually *be* bogus.

Here are some tips on how to get your patch accepted:

2.1. Do send patches, don't send whole archives or files

If your change includes a new file that doesn't exist in the code, then of course you have to send the whole file. But if you're modifying an already-existing file, don't send the whole file. Send a diff instead; specifically, send the output of the diff(1) command run to compare the baseline distributed version against your modified version.

The diff command and its dual, patch(1) (which automatically applies a diff to a baseline file) are the most basic tools of open-source development. Diffs are better than whole files because the developer you're sending a patch to may have changed the baseline version since you got your copy. By sending him a diff you save him the effort of separating your changes from his; you show respect for his time.

2.2. Send patches against the current version of the code.

It is both counterproductive and rude to send a maintainer patches against the code as it existed several releases ago, and expect him to do all the work of determining which changes duplicate things he have since done, versus which things are actually novel in your patch.

As a patch submitter, it is *your* responsibility to track the state of the source and send the maintainer a minimal patch that expresses what you want done to the main-line codebase. That means sending a patch against the current version.

2.3. Don't include patches for generated files.

Before you send your patch, walk through it and delete any patch bands for files in it that are going to be automatically regenerated once he applies the patch and remakes. The classic examples of this error are C files generated by Bison or Flex.

These days the most common mistake of this kind is sending a diff with a huge band that is nothing but changebars between your **configure** script and his. This file is generated by **autoconf**.

This is inconsiderate. It means your recipient is put to the trouble of separating the real content of the patch from a lot of bulky noise. It's a minor error, not as important as some of the things we'll get to further on — but it will count against you.

2.4. Don't send patch bands that just tweak version-control $-symbols.

Some people put special tokens in their source files that are expanded by the version-control system when the file is checked in: the Id construct used by RCS, CVS, and Subversion, for example.

If you're using a local version-control system yourself, your changes may alter these tokens. This isn't really harmful, because when your recipient checks his code back in after applying your patch they'll get re-expanded based on *his* version-control status. But those extra patch bands are noise. They're distracting. It's more considerate not to send them.

This is another minor error. You'll be forgiven for it if you get the big things right. But you want to avoid it anyway.

2.5. Do use -c or -u format, don't use the default (-e) format

The default (-e) format of diff(1) is very brittle. It doesn't include any context, so the patch tool can't cope if any lines have been inserted or deleted in the baseline code since you took the copy you modified.

Getting an -e diff is annoying, and suggests that the sender is either an extreme newbie, careless, or clueless. Most such patches get tossed out without a second thought.

2.6. Do include documentation with your patch

This is very important. If your patch makes a user-visible addition or change to the software's features, *include changes to the appropriate man pages and other documentation files in your patch*. Do *not* assume that the recipient will be happy to document your code for you, or else to have undocumented features lurking in the code.

Documenting your changes well demonstrates some good things. First, it's considerate to the person you are trying to persuade. Second, it shows that you understand the ramifications of your change well enough to explain it to somebody who can't see the code. Third, it demonstrates that you care about the people who will ultimately use the software.

Good documentation is usually the most visible sign of what separates a solid contribution from a quick and dirty hack. If you take the time and care necessary to produce it, you'll find you're already 85% of the way to having your patch accepted with most developers.

2.7. Do include an explanation with your patch

Your patch should include cover notes explaining why you think the patch is necessary or useful. This is explanation directed not to the users of the software but to the maintainer to whom you are sending the patch.

The note can be short — in fact, some of the most effective cover notes I've ever seen just said "See the documentation updates in this patch". But it should show the right attitude.

The right attitude is helpful, respectful of the maintainer's time, quietly confident but unassuming. It's good to display understanding of the code you're patching. It's good to show that you can identify with the maintainer's problems. It's also good to be up front about any risks you perceive in applying the patch. Here are some examples of the sorts of explanatory comments that I like to see in cover notes:

" I've seen two problems with this code, X and Y. I fixed problem X, but I didn't try addressing problem Y because I don't think I understand the part of the code that I believe is involved. "

" Fixed a core dump that can happen when one of the foo inputs is too long. While I was at it, I went looking for similar overflows elsewhere. I found a possible one in blarg.c, near line 666. Are you sure the sender can't generate more than 80 characters per transmission? "

" Have you considered using the Foonly algorithm for this problem? There is a good implementation at <http://www.somesite.com/~jsmith/foonly.html>. "

" This patch solves the immediate problem, but I realize it complicates the memory allocation in an unpleasant way. Works for me, but you should probably test it under heavy load before shipping. "

" This may be featurititis, but I'm sending it anyway. Maybe you'll know a cleaner way to implement the feature. "

2.8. Do include useful comments in your code

Usually as a maintainer, I will want to have strong confidence that I understand your changes before merging them in. This isn't an invariable rule; if you have a track record of good work, with me I may just run a casual eyeball over the changes before checking them in semi-automatically. But everything you can do to help me understand your code and decrease my uncertainty increases your chances that I will accept your patch.

Good comments in your code help me understand it. Bad comments don't.

Here's an example of a bad comment:

```
/* norman newbie fixed this 13 Aug 2009 */
```

This conveys no information. It's nothing but a muddy territorial bootprint you're planting in the middle of my code. If I take your patch (which you've made less likely) I'll almost certainly strip out this comment. If you want a credit, include a patch band for the project `NEWS` or `HISTORY` file. I'll probably take that.

Here's an example of a good comment:

```
/*
 * This conditional needs to be guarded so that crunch_data()
 * never gets passed a NULL pointer.  <norman_newbie@foosite.com>
 */
```

This comment shows that you understand not only my code but the kind of information that I need to have confidence in your changes. This kind of comment *gives* me confidence in your changes.

2.9. Just one bugfix or new feature per patch.

Don't gather various bugfixes, new features or other stuff and send them as one single diff file. Instead, build a single diff for every bugfix or feature. Every single diff should be generated using the current version of the code and should not depend on any other patch you or someone else sent before.

This helps the maintainer to read and understand your code, and he can decide whether he wants to accept or abadon this single bugfix or feature. For example, if you add a feature garanting full root access for everyone because it's useful in your special setting, the maintainer will probably not accept this part of your patch. When you send a single diff file confounding this root access feature along with some bugfixes and other useful things, you have a good chance the maintainer will ignore your complete patch.

With each bugfixes and new feature as a single diff file, the maintainer could e.g. include your bugfixes in a few minutes and take later a closer look to your new features or security issues.

Patches depending on other patches raise similar problems. If the maintainer rejects your basic patch, he won't be able to apply the others. If you can't avoid such dependencies, offer to the maintainer that you will bundle a patch with the sections or features he wants to add.

3. Good project- and archive- naming practice

As the load on maintainers of archives like Metalab, the PSA site and CPAN increases, there is an increasing trend for submissions to be processed partly or wholly by programs (rather than entirely by a human).

This makes it more important for project and archive-file names to fit regular patterns that computer programs can parse and understand.

3.1. Use GNU-style names with a stem and major.minor.patch numbering.

It's helpful to everybody if your archive files all have GNU-like names — all-lower-case alphanumeric stem prefix, followed by a dash, followed by a version number, extension, and other suffixes.

Let's suppose you have a project you call `foobar' at version 1, release 2, level 3. If it's got just one archive part (presumably the sources), here's what its names should look:

foobar-1.2.3.tar.gz
: The source archive

foobar.lsm
: The LSM file (assuming you're submitting to Metalab).

Please *don't* use these:

foobar123.tar.gz
: This looks to many programs like an archive for a project called`foobar123' with no version number.

foobar1.2.3.tar.gz
: This looks to many programs like an archive for a project called `foobar1' at version 2.3.

foobar-v1.2.3.tar.gz
: Many programs think this goes with a project called `foobar-v1'.

foo_bar-1.2.3.tar.gz
: The underscore is hard for people to speak, type, and remember.

FooBar-1.2.3.tar.gz
: Unless you *like* looking like a marketing weenie. This is also hard for people to speak, type, and remember.

If you have to differentiate between source and binary archives, or between different kinds of binary, or express some kind of build option in the file name, please treat that as a file extension to go *after* the version number. That is, please do this:

foobar-1.2.3.src.tar.gz
: sources

foobar-1.2.3.bin.tar.gz
: binaries, type not specified

foobar-1.2.3.bin.ELF.tar.gz
: ELF binaries

foobar-1.2.3.bin.ELF.static.tar.gz
: ELF binaries statically linked

foobar-1.2.3.bin.SPARC.tar.gz

SPARC binaries

Please *don't* use names like `foobar-ELF-1.2.3.tar.gz', because programs have a hard time telling type infixes (like `-ELF') from the stem.

A good general form of name has these parts in order:

1. project prefix
2. dash
3. version number
4. dot
5. "src" or "bin" (optional)
6. dot or dash (dot preferred)
7. binary type and options (optional)
8. archiving and compression extensions

3.2. But respect local conventions where appropriate

Some projects and communities have well-defined conventions for names and version numbers that aren't necessarily compatible with the above advice. For instance, Apache modules are generally named like mod_foo, and have both their own version number and the version of Apache with which they work. Likewise, Perl modules have version numbers that can be treated as floating point numbers (e.g., you might see 1.303 rather than 1.3.3), and the distributions are generally named Foo-Bar-1.303.tar.gz for version 1.303 of module Foo::Bar. (Perl itself, on the other hand, switched to using the conventions described in this document in late 1999.)

Look for and respect the conventions of specialized communities and developers; for general use, follow the above guidelines.

3.3. Try hard to choose a name prefix that is unique and easy to type

The stem prefix should be common to all a project's files, and it should be easy to read, type, and remember. So please don't use underscores. And don't capitalize or BiCapitalize without extremely good reason — it messes up the natural human-eyeball search order, among other things.

It confuses people when two different projects have the same stem name. So try to check for collisions before your first release. Google is your friend — and if the stem you're thinking about gewts lots of Google hits, that in itself may be sufficient reason to think up a different one. Two more good places to check are the index file of Metalab and the appindex at Freshmeat. Another good place to check is SourceForge; do a name search there.

4. Good licensing and copyright practice: the theory

The license you choose defines the social contract you wish to set up among your co-developers and users. The copyright you put on the software will function mainly as a legal assertion of your right to set license terms on the software and derivative works of the software.

4.1. Open source and copyrights

Anything that is not public domain has a copyright, possibly more than one. Under the Berne Convention (which has been U.S. law since 1978), the copyright does not have to be explicit. That is, the authors of a work hold copyright even if there is no copyright notice.

Who counts as an author can be very complicated, especially for software that has been worked on by many hands. This is why licenses are important. By setting out the terms under which material can be used, they grant rights to the users that protect them from arbitrary actions by the copyright holders.

In proprietary software, the license terms are designed to protect the copyright. They're a way of granting a few rights to users while reserving as much legal territory is possible for the owner (the copyright holder). The copyright holder is very important, and the license logic so restrictive that the exact technicalities of the license terms are usually unimportant.

In open-source software, the situation is usually the exact opposite; the copyright exists to protect the license. The only rights the copyright holder always keeps are to enforce the license. Otherwise, only a few rights are reserved and most choices pass to the user. In particular, the copyright holder cannot change the terms on a copy you already have. Therefore, in open-source software the copyright holder is almost irrelevant — but the license terms are very important.

Normally the copyright holder of a project is the current project leader or sponsoring organization. Transfer of the project to a new leader is often signaled by changing the copyright holder. However, this is not a hard and fast rule; many open-source projects have multiple copyright holders, and there is no instance on record of this leading to legal problems.

Some projects choose to assign copyright to the Free Software Foundation, on the theory that it has an interest in defending open source and lawyers available to do it.

4.2. What qualifies as open source

For licensing purposes, we can distinguish several different kinds of rights that a license may convey. Rights to *copy and redistribute*, rights to *use*, rights to *modify for personal use*, and rights to *redistribute modified copies*. A license may restrict or attach conditions to any of these rights.

The Open Source Initiative is the result of a great deal of thought about what makes software "open source" or (in older terminology) "free software". Its constraints on licensing require that:

1. An unlimited right to copy be granted.
2. An unlimited right to use be granted.
3. An unlimited right to modify for personal use be granted.

The guidelines prohibit restrictions on redistribution of modified binaries; this meets the needs of software distributors, who need to be able to ship working code without encumbrance. It allows authors to require that modified sources be redistributed as pristine sources plus patches, thus establishing the author's intentions and an "audit trail" of any changes by others.

The OSD is the legal definition of the `OSI Certified Open Source' certification mark, and as good a definition of "free software" as anyone has ever come up with. All of the standard licenses (MIT, BSD, Artistic, and GPL/LGPL) meet it (though some, like GPL, have other restrictions which you should understand before choosing it).

Note that licenses which allow noncommercial use only do *not* qualify as open-source licenses, even if they are decorated with "GPL" or some other standard license. They discriminate against particular occupations, persons, and groups. They make life too complicated for CD-ROM distributors and others trying to spread open-source software commercially.

5. Good licensing and copyright practice: the practice

Here's how to translate the theory above into practice:

5.1. Make yourself or the FSF the copyright holder

In some cases, if you have a sponsoring organization behind you with lawyers, you might wish to give copyright to that organization.

5.2. Use a license conformant to the Open Source Definition

The Open Source Definition is the community gold standard for licenses. The OSD is not a license itself; rather, it defines a minimum set of rights that a license must guarantee in order to be considered an open-source license. The OSD, and supporting materials, may be found at the web site of the Open Source Initiative.

5.3. Don't write your own license if you can possibly avoid it.

The widely-known OSD-conformant licenses have well-established interpretive traditions. Developers (and, to the extent they care, users) know what they imply, and have a reasonable take on the risks and tradeoffs they involve. Therefore, use one of the standard licenses carried on the OSI site if at all possible.

If you must write your own license, be sure to have it certified by OSI. This will avoid a lot of argument and overhead. Unless you've been through it, you have no idea how nasty a licensing flamewar can get; people become passionate because the licenses are regarded as almost-sacred covenants touching the core values of the open-source community.

Furthermore, the presence of an established interpretive tradition may prove important if your license is ever tested in court. At time of writing (early 2002) there is no case law either supporting or invalidating any open-source license. However, it is a legal doctrine (at least in the U.S., and probably in other common-law countries such as England and the rest of the British Commonwealth) that courts are supposed to interpret licenses and contracts according to the expectations and practices of the community in which they originated.

5.4. Make your license visible in a standard place.

As larger and larger volumes of open-source software are deployed, the problem of auditing these valumes for which licenses cover them become nontrivial — in fact, it becomes larger than an unaided human being can perform. Therefore, it is valuable to have conventions that support mechanized querying of the licensing information. Fortunately, existing community practise already tends in this direction.

As a beginning, the license information for your software should live in a file named either COPYING or LICENSE in the top-level directory of your source distribution. If a single license applies to the entire distribution, that file should include a copy of the license. If multiple licenses apply, that file should list the applicable licenses and indicate to which files and subdirectories they apply.

6. Good development practice

Most of these are concerned with ensuring portability across all POSIX and POSIX-like systems. Being widely portable is not just a worthy form of professionalism and hackerly politeness, it's valuable insurance against future changes in Linux.

Finally, other people *will* try to build your code on non-Linux systems; portability reduces the amount of support email you receive.

6.1. Choose the most portable language you can

Choose a development language which minimizes the differences of the underlying environments in which it will run. C/C++ programs are likely to be more portable than Fortran. Java is preferable to C/C++, and a high-level scripting language such as Perl, Python or Ruby is the best choice of all (since scripting languages have only one cross-platform implementation).

Scripting languages that qualify include Python, Perl, Tcl, Emacs Lisp, and PHP. Historic Bourne shell (/bin/sh) does *not* qualify; there are too many different implementations with subtle idiosyncrasies, and the shell environment is subject to disruption by user customizations such as shell aliases.

If you choose a compiled language such as C/C++, do not use compiler extensions (such as allocating variable-length arrays on the stack, or indirecting through void pointers). Regularly build using as many different compilers and test machines as you can.

6.2. Don't rely on proprietary code

Don't rely on proprietary languages, libraries, or other code. In the open-source community this is considered rude. Open-source developers don't trust code for which they can't review the source.

6.3. Build systems

A significant advantage of open source distributions is they allow each source package to adapt at compile-time to the environment it finds. One of the choices you need to make is of a "build system", the toolkit you (and other people) will rely on to transform your source into executables.

One thing your build script cannot do is ask the user for system information at compile-time. The user installing the package will not necessarily know the answers to your questions, so this approach is doomed from the start. Your software, calling the build system tools, must be able to determine for itself any information that it may need at compile- or install-time.

Community notions of best practice in build systems are now (rearly 2010) in a starte of some flux.

6.3.1. GNU autotools: fading but still standard

Previous versions of this HOWTO urged using GNU autotools to handle portability issues, do system-configuration probes, and tailor your makefiles. This is still the standard and most popular approach, but it is becoming increasingly problematic because GNU autotools is showing its age. Autotools was always a pile of crocks upon hacks upon kluges, implemented in a messy mixture of languages and with some serious design flaws. The resulting mess was tolerable for many years, but as projects become more complex it is increasingly failing.

Still, people building from C sources expect to be able to type "configure; make; make install" and get a clean build. Supposing you choose a non-autotools system, you will probably want to emulate this behavior (which should be easy).

There is a good tutorial on autotools here.

6.3.2. SCons: leading a crowded field

The race to replace autotools does not yet have a winner, but it has a front runner: SCons. SCons abolishes makefiles; it combines the "configure" and "make" parts of the autotools build sequence into one step. It offers cross-platfoem builds with a single recipe on Unix/Linux, Maoc OS X, and Windows. It is written in Python, is extensible in Python, and is to some extent riding the increasing popularity of that language.

6.3.3. CMake and others

SCons is still a minority choice, and has stiff competition from several others, of which CMake and WAF are probably the most prominant. Fairly even-handed cross-comparisons, considering the source, can be found on the SCons wiki.

6.4. Test your code before release

A good test suite allows the team to buy inexpensive hardware for testing and then easily run regression tests before releases. Create a strong, usable test framework so that you can incrementally add tests to your software without having to train developers up in the intricacies of the test suite.

Distributing the test suite allows the community of users to test their ports before contributing them back to the group.

Encourage your developers to use a wide variety of platforms as their desktop and test machines so that code is continuously being tested for portability flaws as part of normal development.

6.5. Sanity-check your code before release

If you're writing C/C++ using GCC, test-compile with -Wall and clean up all warning messages before each release. Compile your code with every compiler you can find — different compilers often find different problems. Specifically, compile your software on true 64-bit machine. Underlying data types can change on 64-bit machines, and you will often find new problems there. Find a UNIX vendor's system and run the lint utility over your software.

Run tools that for memory leaks and other run-time errors; Electric Fence and Valgrind are two good ones available in open source.

For Python projects, the PyChecker program can be a useful check. It's not out of beta yet, but nevertheless often catches nontrivial errors.

If you're writing Perl, check your code with perl -c (and maybe -T, if applicable). Use perl -w and 'use strict' religiously. (See the Perl documentation for further discussion.)

6.6. Sanity-check your documentation and READMEs before release

Spell-check your documentation, README files and error messages in your software. Sloppy code, code that produces warning messages when compiled, and spelling errors in README files or error messages, leads users to believe the engineering behind it is also haphazard and sloppy.

6.7. Recommended C/C++ portability practices

If you are writing C, feel free to use the full ANSI features. Specifically, do use function prototypes, which will help you spot cross-module inconsistencies. The old-style K&R compilers are history.

Do not assume compiler-specific features such as the GCC "-pipe" option or nested functions are available. These will come around and bite you the second somebody ports to a non-Linux, non-GCC system.

Code required for portability should be isolated to a single area and a single set of source files (for example, an "os" subdirectory). Compiler, library and operating system interfaces with portability issues should be abstracted to files in this directory. This includes variables such as "errno", library interfaces such as "malloc", and operating system interfaces such as "mmap".

Portability layers make it easier to do new software ports. It is often the case that no member of the development team knows the porting platform (for example, there are literally hundreds of different embedded operating systems, and nobody knows any significant fraction of them). By creating a separate portability layer it is possible for someone who knows the port platform to port your software without having to understand it.

Portability layers simplify applications. Software rarely needs the full functionality of more complex system calls such as mmap or stat, and programmers commonly configure such complex interfaces incorrectly. A portability layer with abstracted interfaces ("__file_exists" instead of a call to stat) allows you to export only the limited, necessary functionality from the system, simplifying the code in your application.

Always write your portability layer to select based on a feature, never based on a platform. Trying to create a separate portability layer for each supported platform results in a multiple update problem maintenance nightmare. A "platform" is always selected on at least two axes: the compiler and the library/operating system release. In some cases there are three axes, as when Linux vendors select a C library independently of the operating system release. With M vendors, N compilers and O operating system releases, the number of "platforms" quickly scales out of reach of any but the largest development teams. By using language and systems standards such as ANSI and POSIX 1003.1, the set of features is relatively constrained.

Portability choices can be made along either lines of code or compiled files. It doesn't make a difference if you select alternate lines of code on a platform, or one of a few different files. A rule of thumb is to move portability code for different platforms into separate files when the implementations diverged significantly (shared memory mapping on UNIX vs. Windows), and leave portability code in a single file when the differences are minimal (using gettimeofday, clock_gettime, ftime or time to find out the current time-of-day).

Avoid using complex types such as "off_t" and "size_t". They vary in size from system to system, especially on 64-bit systems. Limit your usage of "off_t" to the portability layer, and your usage of "size_t" to mean only the length of a string in memory, and nothing else.

Never step on the namespace of any other part of the system, (including file names, error return values and function names). Where the namespace is shared, document the portion of the namespace that you use.

Choose a coding standard. The debate over the choice of standard can go on forever — regardless, it is too difficult and expensive to maintain software built using multiple coding standards, and so some coding standard must be chosen. Enforce your coding standard ruthlessly, as consistency and cleanliness of the code are of the highest priority; the details of the coding standard itself are a distant second.

7. Good distribution-making practice

These guidelines describe how your distribution should look when someone downloads, retrieves and unpacks it.

7.1. Make sure tarballs always unpack into a single new directory

The single most annoying mistake newbie developers make is to build tarballs that unpack the files and directories in the distribution into the current directory, potentially stepping on files already located there. *Never do this!*

Instead, make sure your archive files all have a common directory part named after the project, so they will unpack into a single top-level directory directly *beneath* the current one.

Here's a makefile trick that, assuming your distribution directory is named `foobar' and SRC contains a list of your distribution files, accomplishes this. SRC may also contain names of subdirectories to be included whole.

```
foobar-$(VERS).tar.gz:
        @find $(SRC) -type f | sed s:^:foobar-$(VERS)/: >MANIFEST
        @(cd ..; ln -s foobar foobar-$(VERS))
        (cd ..; tar -czvf foobar/foobar-$(VERS).tar.gz `cat foobar/MANIFEST`)
        @(cd ..; rm foobar-$(VERS))
```

7.2. Have a README

Have a file called `README` or `READ.ME` that is a roadmap of your source distribution. By ancient convention, this is the first file intrepid explorers will read after unpacking the source.

Good things to have in the README include:

1. A brief description of the project.
2. A pointer to the project website (if it has one)
3. Notes on the developer's build environment and potential portability problems.
4. A roadmap describing important files and subdirectories.
5. Either build/installation instructions or a pointer to a file containing same (usually `INSTALL`).
6. Either a maintainers/credits list or a pointer to a file containing same (usually `CREDITS`).
7. Either recent project news or a pointer to a file containing same (usually `NEWS`).

7.3. Respect and follow standard file naming practices

Before even looking at the README, your intrepid explorer will have scanned the filenames in the top-level directory of your unpacked distribution. Those names can themselves convey information. By adhering to certain standard naming practices, you can give the explorer valuable clues about what to look in next.

Here are some standard top-level file names and what they mean. Not every distribution needs all of these.

README or READ.ME
: the roadmap file, to be read first

INSTALL
: configuration, build, and installation instructions

AUTHORS
: list of project contributers.

  An older, still-acceptable convention for this is to name it CREDITS

NEWS
: recent project news

HISTORY
: project history

COPYING
: project license terms (GNU convention)

LICENSE
: project license terms

MANIFEST
: list of files in the distribution

FAQ
: plain-text Frequently-Asked-Questions document for the project

TAGS
: generated tag file for use by Emacs or vi

Note the overall convention that filenames with all-caps names are human-readable metainformation about the package, rather than build components (TAGS is an exception to the first, but not to the second).

Having a FAQ can save you a lot of grief. When a question about the project comes up often, put it in the FAQ; then direct users to read the FAQ before sending questions or bug reports. A well-nurtured FAQ can decrease the support burden on the project maintainers by an order of magnitude or more.

Having a HISTORY or NEWS file with timestamps in it for each release is valuable. Among other things, it may help establish prior art if you are ever hit with a patent-infringement lawsuit (this hasn't happened to anyone yet, but best to be prepared).

7.4. Design for Upgradability

Your software will change over time as you put out new releases. Some of these changes will not be backward-compatible. Accordingly, you should give serious thought to designing your installation layouts so that multiple installed versions of your code can coexist on the same system. This is especially important for libraries — you can't count on all your client programs to upgrade in lockstep with your API changes.

The Emacs, Python, and Qt projects have a good convention for handling this; version-numbered directories. Here's how an installed Qt library hierarchy looks (${ver} is the version number):

```
/usr/lib/qt
/usr/lib/qt-${ver}
/usr/lib/qt-${ver}/bin          # Where you find moc
/usr/lib/qt-${ver}/lib          # Where you find .so
/usr/lib/qt-${ver}/include      # Where you find header files
```

With this organization, you can have multiple versions coexisting. Client programs have to specify the library version they want, but that's a small price to pay for not having the interfaces break on them.

7.5. Provide checksums

Provide checksums with your binaries (tarballs, RPMs, etc.). This will allow people to verify that they haven't been corrupted or had Trojan-horse code inserted in them.

While there are several commands you can use for this purpose (such as **sum** and **cksum**) it is best to use a cryptographically-secure hash function. The GPG package provides this capability via the `—detach-sign` option; so does the GNU command **md5sum**.

For each binary you ship, your project web page should list the checksum and the command you used to generate it.

8. Good documentation practice

The most important good documentation practice is to actually write some! Too many programmers omit this. But here are two good reasons to do it:

1. *Your documentation can be your design document.* The best time to write it is before you type a single line of code, while you're thinking out what you want to do. You'll find that the process of describing the way you want your program to work in natural language focuses your mind on the high-level questions about what it should do and how it should work. This may save you a lot of effort later.
2. *Your documentation is an advertisement for the quality of your code.* Many people take poor, scanty, or illiterate documentation for a program as a sign that the programmer is sloppy or careless of potential users' needs. Good documentation, on the other hand, conveys a message of intelligence and professionalism. If your program has to compete with other programs, better make sure your documentation is at least as good as theirs lest potential users write you off without a second look.

This HOWTO wouldn't be the place for a course on technical writing even if that were practical. So we'll focus here on the formats and tools available for composing and rendering documentation.

Though Unix and the open-source community have a long tradition of hosting powerful document-formatting tools, the plethora of different formats has meant that documentation has tended to be fragmented and difficult for users to browse or index in a coherent way. We'll summarize the uses, strengths, and weaknesses of the common documentation formats. Then we'll make some recommendations for good practice.

8.1. Documentation formats

Here are the documentation markup formats now in widespread use among open-source developers. When we speak of "presentation" markup, we mean markup that controls the document's appearance explicitly (such as a font change). When we speak of "structural" markup, we mean markup that describes the logical structure of the document (like a section break or emphasis tag.) And when we speak of "indexing", we mean the process of extracting from a collection of documents a searchable collection of topic pointers that users can employ to reliably find material of interest across the entire collection.

man pages

The most most common format, inherited from Unix, a primitive form of presentation markup. man(1) command provides a pager and a stone-age search facility. No support for images or hyperlinks or indexing. Renders to Postscript for printing fairly well. Doesn't render to HTML at all well (essentially as flat text). Tools are preinstalled on all Linux systems.

Man page format is not bad for command summaries or short reference documents intended to jog the memory of an experienced user. It starts to creak under the strain for programs with complex interfaces and many options, and collapses entirely if you need to maintain a set of documents with rich cross-references (the markup has only weak and normally unused support for hyperlinks).

HTML

Increasingly common since the Web exploded in 1993-1994. Markup is partly structural, mostly presentation. Browseable through any web browser. Good support for images and hyperlinks. Limited built-in facilities for indexing, but good indexing and search-engine technologies exist and are widely deployed. Renders to Postscript for printing pretty well. HTML tools are now universally available.

HTML is very flexible and suitable for many kinds of documentation. Actually, it's *too* flexible; it shares with man page format the problem that it's hard to index automatically because a lot of the markup describes presentation rather than document structure.

Texinfo
: Texinfo is the documentation format used by the Free Software Foundation. It's a set of macros on top of the powerful TeX formatting engine. Mostly structural, partly presentation. Browseable through Emacs or a standalone **info** program. Good support for hyperlinks, none for images. Good indexing for both print and on-line forms; when you install a Texinfo document, a pointer to it is automatically added to a browsable "dir" document listing all the Texinfo documents on your system. Renders to excellent Postscript and useable HTML. Texinfo tools are preinstalled on most Linux systems, and available at the Free Software Foundation website.

  Texinfo is a good design, quite usable for typesetting books as well as small on-line documents, but like HTML it's a sort of amphibian — the markup is part structural, part presentation, and the presentation part creates problems for rendering.

DocBook
: DocBook is a large, elaborate markup format based on SGML (more recent versions on XML). Unlike the other formats described here it is entirely structural with no presentation markup. Excellent support for images and hyperlinks. Good support for indexing. Renders well to HTML, acceptably to Postscript for printing (quality is improving as the tools evolve). Tools and documentation are available at the DocBook website.

  DocBook is excellent for large, complex documents; it was designed specifically to support technical manuals and rendering them in multiple output formats. Its main drawback is its verbosity. Fortunately, good tools and introductory-level documentation are now available; see the DocBook Demystification HOWTO for an introduction.

asciidoc
: The one serious drawback of DocBook is that its markup is rather heavyweight and obtrusive. A clever recent workaround is AsciiDOC. This tool is a front end to DocBook with a much simpler and more natural input syntax. Users don't nered to be aware od DocBook at all, but still ret nearly the fill power of those tools.

  AsciiDOC (often referred to by the all-lower-case name of the formatter it ships) has been seeing very rapid uptake recently by projects which had previously moved to DocBook.

8.2. Good practice recommendations

Documentation practice has been changing since 2000, when some key open-source project groups (including the Linux kernel project, GNOME, KDE, the Free Software Foundation, and the Linux Documentation Project) agreed on an approach more web-friendly than traditional Unix's print-oriented tools. Today's best practice, since the XML-DocBook toolchain reached production status in mid-2001, is this:

1. Maintain your document masters in either XML-DocBook or asciidoc. Even your man pages can be DocBook `RefEntry` documents. There is a very good HOWTO on writing manual pages that explains the sections and organization your users will expect to see.
2. Ship the XML or asciidoc masters. Also, in case your users' systems don't have xmlto(1) (standard on all Red Hat distributions since 7.3), ship the troff sources that you get by running conversions on your masters. Your software distribution's installation procedure should install those in the normal way, but direct people to the XML/asciidoc files if they want to write documentation patches.

It's easy to tell make(1) to keep the generated man files up to date. Just do something like this in your makefile:

```
foo.1: foo.xml
        xmlto man foo.xml
```

If you're using asciidoc, something like this should serve:

```
foo.1: foo.txt
        asciidoc --backend=docbook foo.txt
        xmlto man foo.xml
```

3. Generate XHTML from your masters (with **xmlto xhtml**, or directly using **asciidoc**) and make it available from your project's web page, where people can browse it in order to decide whether to download your code and join your project.

For converting legacy documentation in troff formats to DocBook, check out doclifter. If you're unwilling to move from using man sources as a master format, at least try to clean them up so doclifter can lift them to XML automatically.

9. Good communication practice

Your software and documentation won't do the world much good if nobody but you knows it exists. Also, developing a visible presence for the project on the Internet will assist you in recruiting users and co-developers. Here are the standard ways to do that.

9.1. Announce to c.o.l.a and Freshmeat

Announce new releases to comp.os.linux.announce. Besides being widely read itself, this group is a major feeder for web-based what's-new sites like Freshmeat.

9.2. Announce to a relevant topic newsgroup

Find USENET topics group directly relevant to your application, and announce there as well. Post only where the *function* of the code is relevant, and exercise restraint.

If (for example) you are releasing a program written in Perl that queries IMAP servers, you should certainly post to comp.mail.imap. But you should probably not post to comp.lang.perl unless the program is also an instructive example of cutting-edge Perl techniques.

Your announcement should include the URL of a project website.

9.3. Have a website

If you intend try to build any substantial user or developer community around your project, it should have a website. Standard things to have on the website include:

- The project charter (why it exists, who the audience is, etc).
- Download links for the project sources.
- Instructions on how to join the project mailing list(s).
- A FAQ (Frequently Asked Questions) list.
- HTMLized versions of the project documentation
- Links to related and/or competing projects.

Some project sites even have URLs for anonymous access to the master source tree.

9.4. Host project mailing lists

It's standard practice to have a private development list through which project collaborators can communicate and exchange patches. You may also want to have an announcements list for people who want to be kept informed of the project's process.

If you are running a project named `foo'. your developer list might be foo-dev or foo-friends; your announcement list might be foo-announce.

9.5. Release to major archives

Since it was launched in fall 1999, SourceForge has exploded in popularity. It is not just an archive and distribution site, though you can use it that way. It is an entire free project-hosting service that tries to offer a complete set of tools for open-source development groups — web and archive space, mailing lists, bug-tracking, chat forums, CVS repositories, and other services.

Other important locations include:

- the Python Software Activity site (for software written in Python).
- the CPAN, the Comprehensive Perl Archive Network, (for software written in Perl).

10. Good project-management practice

Managing a project well when all the participants are volunteers presents some unique challenges. This is too large a topic to cover in a HOWTO. Fortunately, there are some useful white papers available that will help you understand the major issues.

For discussion of basic development organization and the release-early-release-often `bazaar mode', see The Cathedral and the Bazaar.

For discussion of motivational psychology, community customs, and conflict resolution, see Homesteading the Noosphere.

For discussion of economics and appropriate business models, see The Magic Cauldron.

These papers are not the last word on open-source development. But they were the first serious analyses to be written, and have yet to be superseded (though the author hopes they will be someday).