Free Software Project Management HOWTO

Benjamin "Mako" Hill

mako@debian.org

Revision History

Revision v0.3.2	15 April 2002	Revised by: bch
Revision v0.3.1	18 June 2001	Revised by: bch
Revision v0.3	5 May 2001	Revised by: bch
Revision v0.2.1	10 April 2001	Revised by: bch
Revision v0.2	8 April 2001	Revised by: bch
Revision v0.01	27 March 2001	Revised by: bch
Initial Release		

This HOWTO is designed for people with experience in programming and some skills in managing a software project but who are new to the world of free software. This document is meant to act as a guide to the non–technical aspects of free software project management and was written to be a crash course in the people skills that aren't taught to commercial coders but that can make or break a free software project.

Table of Contents

Table of Contents

1. Introduction

Skimming through freshmeat.net provides mountains of reasons for this HOWTO's existence−−the Internet is littered with excellently written and useful programs that have faded away into the universe of free software forgottenness. This dismal scene made me ask myself, "Why?"

This HOWTO tries to do a lot of things (probably too many), but it can't answer that question and won't attempt it. What this HOWTO will attempt to do is give your Free Software project a fighting chance−−an edge. If you write a piece of crap that no one is interested in, you can read this HOWTO until you can recite it in your sleep and your project will probably fail. Then again, you can write a beautiful, relevant piece of software and follow every instruction in this HOWTO and your software may still not make it. Sometimes life is like that. However, I'll go out a limb and say that if you write a great, relevant pieces of software and ignore the advise in this HOWTO, you'll probably fail *more often*.

A lot of the information in this HOWTO is best called common sense. Of course, as any debate on interfaces will prove, what is common sense to some programmers proves totally unintuitive to others. After explaining bits and pieces of this HOWTO to Free Software developers on several occasions, I realized that writing this HOWTO might provide a useful resource and a forum for programmers to share ideas about what has and has not worked for them.

As anyone involved in any of what seems like an unending parade of ridiculous intellectual property clashes will attest to, a little bit of legalese proves important.

1.1. Copyright Information

1.2. Disclaimer

No liability for the contents of this documents can be accepted. Use the concepts, examples and other content at your own risk. As this is a new edition of this document, there may be errors and inaccuracies, that may of course be damaging to your project (and potentially your system). Proceed with caution, and although this is highly unlikely, the author(s) does not take any responsibility for that.

All copyrights are held by their by their respective owners, unless specifically noted otherwise. Use of a term in this document should not be regarded as affecting the validity of any trademark or service mark.

Naming of particular products or brands should not be seen as endorsements.

1.3. New Versions

This version is the part of the third pre–release cycle of this HOWTO. It is written to be released to developers for critique and brainstorming. Please keep in mind that this version of the HOWTO is still in an infant stage and will continue to be revised extensively.

The latest version number of this document should always be listed on the projects homepage hosted by yukidoke.org.

The newest version of this HOWTO will always be made available at the same website, in a variety of formats:

- HTML.
- HTML (single page).
- plain text.
- Compressed postscript.
- Compressed SGML source.

1.4. Credits

In this version I have the pleasure of acknowledging:

Fellow Debian developer Martin Michlmayr and Vivek Venugopalan who sent me information and links to extremely interesting articles. I've added both to the bibliography and I've added information from each into the HOWTO. Thanks to Andrew Shugg who pointed out several errors in the document. Also, a big thanks to Sung Wook Her (AKA RedBaron) who is doing the first translation of the HOWTO into Korean. I've been happy to see that people have enjoyed and benefited from the HOWTO so far.

Older thanks that I don't want to take out yet include: Josh Crawford, Andy King, and Jaime Davila who all read through this in entirety and gave me feedback that has helped me make changes and improvements to this document. I can't thank you guys enough for your help. An extra "Thank You" goes to Andy King who who read through this several times and submitted patches to make life easier for me.

Karl Fogel, the author of *Open Source Development with CVS* published by the Coriolis Open Press. Large parts of his book are available on the web. 225 pages of the book are available under the GPL and constitute the best tutorial on CVS I've ever seen. The rest of the book covers, "the challenges and philosophical issues inherent in running an Open Source project using CVS." The book does a good job of covering some of the subjects brought up in this HOWTO and much more. The book's website has information on ordering the book and provides several translations of the chapters on CVS. If you are seriously interested in running a Free Software project, you want this book. I tried to mention Fogel in sections of this HOWTO where I knew I was borrowing directly from his ideas. If I missed any, I'm sorry. I'll try and have those fixed in future versions.

Karl Fogel can be reached at `<kfogel (at) red-bean (dot) com>`

Also providing support material, and inspiration for this HOWTO is Eric S. Raymond for his prolific, consistent, and carefully crafted arguments and Lawrence Lessig for reminding me of the importance of Free Software. Additionally, I want to thank every user and developer involved with the Debian Project. The project has provided me with a home, a place to practice free software advocacy, a place to make a difference, a place to learn from those who have been involved with the movement much longer than I, and proof of a free software project that definitely, definitely works.

Above all, I want to thank *Richard Stallman* for his work at the Free Software Foundation and for never giving up. Stallman provides and articulates the philosophical basis that attracts me to free software and that drives me toward writing a document to make sure it succeeds. RMS can always be emailed at `<rms (at) gnu (dot) org>`.

1.5. Feedback

Feedback is always and most certainly welcome for this document. Without your submissions and input, this document wouldn't exist. Do you feel that something is missing? Don't hesitate to contact me to have me write a chapter, section, or subsection or to write one yourself. I want this document to be a product of the Free Software development process that it heralds and I believe that its ultimate success will be rooted in its ability to do this. Please send your additions, comments, and criticisms to the following email address: `<mako@debian.org>`.

1.6. Translations

I know that not everyone speaks English. Translations are nice and I'd love for this HOWTO to gain the kind of international reach afforded by translated versions.

I've been contacted by a reader who promises a translation into Korean. However, this HOWTO is still young and other than the promise of Korean, English is all that is currently available. If you would like to help with or do a translation, you will gain my utmost respect and admiration and you'll get to be part of a cool process. If you are at all interested, please don't hesitate to contact me at: `<mako@debian.org>`.

2. Starting a Project

With very little argument, the beginning is the most difficult period in a project's life to do successful free software project management. Laying a firm foundation will determine whether your project flourishes or withers away and dies. It is also the subject that is of most immediate interest to anyone reading this document as a tutorial.

Starting a project involves a dilemma that you as a developer must try and deal with: no potential user for your program is interested in a program that doesn't work, while the development process that you want to employ holds involvement of users as imperative.

It is in these dangerous initial moments that anyone working to start a free software project must try and strike a balance along these lines. One of the most important ways that someone trying to start a project can work toward this balance is by establishing a solid framework for the development process through some of the suggestions mentioned in this section.

2.1. Choosing a Project

If you are reading this document, there's a good chance you already have an idea for a project in mind. Chances are also pretty good that it fills a perceived gap by doing something that no other free software project does or by doing something in a way that is unique enough to necessitate a brand new piece of software.

2.1.1. Identify and articulate your idea

Eric S. Raymond writes about how free software projects start in his essay, "The Cathedral and the Bazaar," which comes as required reading for any free software developer. It is available online .

In "The Cathedral and the Bazaar," Raymond tells us that: "every good work of software starts by scratching a developers itch." Raymond's now widely accepted hypothesis is that new free software programs are written, first and foremost, to solve a specific problem facing the developer.

If you have an idea for a program in mind, chances are good that it targets a specific problem or "itch" you want to see scratched. *This idea is the project.* Articulate it clearly. Write it out. Describe the problem you will attack in detail. The success of your project in tackling a particular problem will be tied to your ability to identify that problem clearly early on. Find out exactly what it is that you want your project to do.

Monty Manley articulates the importance of this initial step in an essay, "Managing Projects the Open Source Way." As the next section will show, there is *a lot* of work that needs to be done before software is even ready to be coded. Manley says, "Beginning an OSS project properly means that a developer must, first and foremost, avoid writing code too soon!"

2.1.2. Evaluate your idea

In evaluating your idea, you need to first ask yourself a few questions. This should happen before you move any further through this HOWTO. Ask yourself: *Is the free software development model really the right one for your project?*

Obviously, since the program scratches your itch, you are definitely interested in seeing it implemented in code. But, because one hacker coding in solitude fails to qualify as a free software development effort, you need to ask yourself a second question: *Is anybody else interested?*

Sometimes the answer is a simple "no." If you want to write a set of scripts to sort *your* MP3 collection on *your* machine, *maybe* the free software development model is not the best one to choose. However, if you want to write a set of scripts to sort *anyone's* MP3s, a free software project might fill a useful gap.

Luckily, the Internet is a place so big and so diverse that, chances are, there is someone, somewhere, who shares your interests and who feels the same "itch." It is the fact that there are so many people with so many similar needs and desires that introduces the third major question: *Has somebody already had your idea or a reasonably similar one?*

2.1.2.1. Finding Similar Projects

There are places you can go on the web to try and answer the question above. If you have experience with the free software community, you are probably already familiar with many of these sites. All of the resources listed below offer searching of their databases:

freshmeat.net

> freshmeat.net describes itself as, "the Web's largest index of Linux and Open Source software" and its reputation along these lines is totally unparalleled and unquestioned. If you can't find it on freshmeat, its doubtful that you (or anyone else) will find it at all.

Slashdot

> Slashdot provides "News for Nerds. Stuff that matters," which usually includes discussion of free software, open source, technology, and geek culture news and events. It is not unusual for a particularly sexy development effort to be announced here, so it is definitely worth checking.

SourceForge

> SourceForge houses and facilitates a growing number of open source and free software projects. It is also quickly becoming a nexus and a necessary stop for free software developers. SourceForge's software map and new release pages should be necessary stops before embarking on a new free software project. SourceForge also provides a Code Snippet Library which contains useful reusable chunks of code in an array of languages which can come in useful in any project.

Google and Google's Linux Search

> Google and Google's Linux Search, provides powerful web searches that may reveal people working on similar projects. It is not a catalog of software or news like freshmeat or Slashdot, but it is worth checking to make sure you aren't pouring your effort into a redundant project.

2.1.2.2. Deciding to Proceed

Once you have successfully charted the terrain and have an idea about what kinds of similar free software projects exist, every developer needs to decide whether to proceed with their own project. It is rare that a new project seeks to accomplish a goal that is not at all similar or related to the goal of another project. Anyone starting a new project needs to ask themselves: "Will the new project be duplicating work done by another project? Will the new project be competing for developers with an existing project? Can the goals of the new project be accomplished by adding functionality to an existing project?"

If the answer to any of these questions is "yes," try to contact the developer of the existing project(s) in question and see if he or she might be willing to collaborate with you.

For many developers this may be the single most difficult aspect of free software project management, but it is an essential one. It is easy to become fired up by an idea and get caught up in the momentum and excitement of a new project. It is often extremely difficult to do, but it is important that any free software developer remembers that the best interests of the free software community and the quickest way to accomplish your own project's goals and the goals of similar projects can often be accomplished by *not* starting a new development effort.

2.2. Naming your project

While there are plenty of projects that fail with descriptive names and plenty that succeed without them, I think naming your project is worth giving a bit of thought. Leslie Orchard tackles this issue in an Advogato article. His article is short and definitely worth looking over quickly.

The synopsis is that Orchard recommends you pick a name where, after hearing the name, many users or developers will both:

- Know what the project does.
- Remember it tomorrow.

Humorously, Orchard's project, "Iajitsu," does neither. It is probably unrelated that development has effectively frozen since the article was written.

He makes a good point though. There are companies whose only job is to make names for pieces of software. They make *ridiculous* amount of money doing it and are supposedly worth it. While you probably can't afford a company like this, you can afford to learn from their existence and think a little bit about the name you are giving your project because it *does* matter.

If there is a name you really want but it doesn't fit Orchard's criteria, you can still go ahead. I thought "gnubile" was one of the best I'd heard for a free software project ever and I still talk about it long after I've stopped using the program. However, if you can be flexible on the subject, listen to Orchard's advice. It might help you.

2.3. Licensing your Software

On one (somewhat simplistic) level, the difference between a piece of free software and a piece of propriety software is the license. A license helps you as the developer by protecting your legal rights to have your software distributed under your terms and helps demonstrate to those who wish to help you or your project that they are encouraged to join.

2.3.1. Choosing a license

Any discussion of licenses is also sure to generate at least a small flame war as there are strong feelings that some free software licenses are better than others. This discussion also brings up the question of "Open Source Software" and the debate over the terms "Open Source Software" and "Free Software". However, because I've written the Free Software Project Management HOWTO and not the Open Source Software Project Management HOWTO, my own allegiances in this argument are in the open.

In attempting to reach a middle ground through diplomacy without sacrificing my own philosophy, I will recommend picking any license that conforms to the Debian Free Software Guidelines. Originally compiled by the Debian project under Bruce Perens, the DFSG forms the first version of the Open Source Definition. Examples of free licenses given by the DFSG are the GPL, the BSD, and the Artistic License. As ESR mentions in his his HOWTO[ESRHOWTO], don't write your own license if at all possible. The three licenses I mention all have long interpretive traditions. They are also definitely free software (and can therefore be distributed as part of Debian and in other places that permit the transfer of free software).

Conforming to the definition of free software offered by Richard Stallman in "The Free Software Definition", any of these licenses will uphold, "users' freedom to run, copy, distribute, study, change and improve the software." There are plenty of other licenses that also conform to the DFSG but sticking with a more well−known license will offer the advantage of immediate recognition and understanding. Many people write three or four sentences in a COPYING file and assume that they have written a free software license−−as my long experience with the debian−legal mailing professes, this is very often not the case.

In attempting a more in−depth analysis, I agree with Karl Fogel's description of licenses as falling into two groups: those that are the GPL and those that are not the GPL.

Personally, I license all my software under the GPL. Created and protected by the Free Software Foundation and the GNU Project, the GPL is the license for the Linux kernel, GNOME, Emacs, and the vast majority of GNU/Linux software. It's the obvious choice but I also believe it is a good one. Any BSD fanatic will urge you to remember that there is a viral aspect to the GPL that prevents the mixture of GPL'ed code with non−GPL'ed code. To many people (myself included), this is a benefit, but to some, it is a major drawback.

Many people write three or four sentences in a COPYING file and assume that they have written a free software license−−as my long experience with the debian−legal mailing professes, this is very often not the case. It may not protect you, it may not protect your software, and it may make things very difficult for people that want to use your software but who pay a lot of attention to the subtle legal points of licenses. If you are passionate about a home−brewed license, run it by either people at OSI or the debian−legal mailing list first protect yourself from unanticipated side−effects of your license.

The three major licenses can be found at the following locations:

- The GNU General Public License
- The BSD License
- The Artistic License

In any case, please read through any license before your release your software under it. As the primary developer, you can't afford any license surprises.

2.3.2. The mechanics of licensing

The text of the GPL offers a good description of the mechanics of applying a license to a piece of software. My quick checklist for applying a license includes:

- Make yourself or the FSF the copyright holder for the work. In a few rare cases, you might want to make a sponsoring organization (if it's big and powerful enough) the copyright holder instead. Doing this is as simple as putting the name in the blank when you modify the notice of copyright below. Contrary to popular belief, you don't need to file with any organization. The notice alone is enough to copyright your work.
- If at all possible, attach and distribute a full copy of the license with the source and binary by including a separate file.
- At the top of each source file in your program, attach a notice of copyright and include information on where the full license can be found. The GPL recommends that each file begin with:

```
one line to give the program's name and an idea of what it does.
Copyright (C) yyyy  name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
```

  The GPL goes on to recommend attaching information on methods for contacting you (the author) via email or physical mail.

- The GPL continues and suggests that if your program runs in an interactive mode, you should write the program to output a notice each time it enters interactive mode that includes a message like this one that points to full information about the programs license:

```
Gnomovision version 69, Copyright (C) year name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type `show w'.  This is free software, and you are welcome
to redistribute it under certain conditions; type `show c'
for details.
```

- Finally, it might be helpful to include a "copyright disclaimer" from an employer or a school if you work as a programmer or if it seems like your employer or school might be able to make an argument for ownership of your code later on. These aren't often needed but there are plenty of free software developers who have gotten into trouble and wish they'd asked for one.

2.3.3. Final license warning

Please, please, please, place your software under *some* license. It may not seem important, and to you it may not be, but licenses *are* important. For a piece of software to be included in the Debian GNU/Linux distribution, it must have a license that fits the Debian Free Software Guidelines. If your software has no license, it can not be distributed as a package in Debian until you re–release it under a free license. Please save yourself and others trouble by releasing the first version of your software with a clear license.

2.4. Choosing a Method of Version Numbering

The most important thing about a system of version numbering is that there is one. It may seem pedantic to emphasize this point but you'd be surprised at the number of scripts and small programs that pop up without any version number at all.

The second most important thing about a system of numbering is that the numbers always go up. Automatic version tracking systems and people's sense of order in the universe will fall apart if version numbers don't rise. It doesn't *really* matter if 2.1 is a big jump and 2.0.005 is a small jump but it does matter that 2.1 is more recent than 2.0.005.

Follow these two simple rules and you will not go (too) wrong. Beyond this, the most common technique seems to be the "major level," "minor level," "patch level" version numbering scheme. Whether you are familiar with the name or not, you interact with it all the time. The first number is the major number and it signifies major changes or rewrites. The second number is the minor number and it represents added or tweaked functionality on top of a largely coherent structure. The third number is the patch number and it usually will only refer to releases fixing bugs.

The widespread use of this scheme is why I know the nature and relative degree in the differences between a 2.4.12 release of the Linux kernel and a 2.4.11, 2.2.12, and 1.2.12 without knowing anything about any of the releases.

You can bend or break these rules, and people do. But beware, if you choose to, someone will get annoyed, assume you don't know, and try and educate you, probably not nicely. I always follow this method and I implore you to do so as well.

There are several version numbering systems that are well known, useful, and that might be worth looking into before you release your first version.

Linux kernel version numbering:

> The Linux kernel uses a versioning system where any odd minor version number refers to an development or testing release and any even minor version number refers to a stable version. Think about it for a second. Under this system, 2.1 and 2.3 kernels were and always will be development or testing kernels and 2.0, 2.2. and 2.4 kernels are all production code with a higher degree of stability and more testing.
>
> Whether you plan on having a split development model (as described in Section 3.3) or only one version released at a time, my experience with several free software projects and with the Debian project has taught me that use of Linux's version numbering system is worth taking into consideration. In Debian, *all* minor versions are stable distributions (2.0, 2.1, etc). However, many

people assume that 2.1 is an unstable or development version and continue to use an older version until they get so frustrated with the lack of development progress that they complain and figure the system out. If you never release an odd minor version but only release even ones, nobody is hurt, and less people are confused. It's an idea worth taking into consideration.

Wine version numbering:

Because of the unusual nature of wine's development where the not−emulator is constantly improving but not working toward any immediately achievable goal, wine is released every three weeks. Wine does this by labeling their releases in "Year Month Day" format where each release might be labeled "wine−XXXXXXXX" where the version from January 04, 2000 would be "wine−20000104". For certain projects, "Year Month Day" format can make a lot of sense.

Mozilla milestones:

When one considers Netscape 6 and vendor versions, the mozilla's project development structure is one of the most complex free software models available. The project's version numbering has reflected the unique situation in which it is developed.

Mozilla's version numbering structure has historically been made up of milestones. From the beginning of the mozilla project, the goals of the project in the order and degree to which they were to be achieved were charted out on a series of road maps. Major points and achievements along these road−maps were marked as milestones. Therefore, although Mozilla was built and distributed nightly as "nightly builds," on a day when the goals of a milestone on the road−map had been reached, that particular build was marked as a "milestone release."

While I haven't seen this method employed in any other projects to date, I like the idea and think that it might have value in any testing or development branch of a large application under heavy development.

2.5. Documentation

A huge number of otherwise fantastic free software applications have withered and died because their author was the only person who knew how to use them fully. Even if your program is written primarily for a techno−savvy group of users, documentation is helpful and even necessary for the survival of your project. You will learn later in Section 4.3 that you should always release something that is usable. *A piece of software without documentation is not usable.*

There are lots of different people you should document for and there are lots of ways to document your project. *The importance of documentation in source code to help facilitate development by a large community is vital* but it falls outside the scope of this HOWTO. This being the case, this section deals with useful tactics for user−directed documentation.

A combination of tradition and necessity has resulted in a semi−regular system of documentation in most free software projects that is worth following. Both users and developers expect to be able to get documentation in several ways and it's essential that you provide the information they are seeking in a form they can read if your project is ever going to get off the ground. People have come to expect:

2.5.1. Man pages

Your users will want to be able to type "man yourprojectname" end up with a nicely formatted man page highlighting the basic use of your application. Make sure that before you release your program, you've planned for this.

Man pages are not difficult to write. There is excellent documentation on the man page writing process available through the "The Linux Man–Page–HOWTO" which is available through the Linux Documentation project (LDP) and is written by Jens Schweikhardt. It is available from Schweikhardt's site or from the LDP.

It is also possible to write man pages using DocBook SGML. Because man pages are so simple and the DocBook method relatively new, I have not been able to follow this up but would love help from anyone who can give me more information on how exactly how this is done.

2.5.2. Command line accessible documentation

Most users will expect some basic amount of documentation to be easily available from the command line. For few programs should this type of documentation extend for more than one screen (24 or 25 lines) but it should cover the basic usage, a brief (one or two sentence) description of the program, a list of the commands with explanations, as well as all the major options (also with explanations), plus a pointer to more in–depth documentation for those who need it. The command line documentation for Debian's apt–get serves as an excellent example and a useful model:

```
apt 0.3.19 for i386 compiled on May 12 2000  21:17:27
Usage: apt-get [options] command
       apt-get [options] install pkg1 [pkg2 ...]

apt-get is a simple command line interface for downloading and
installing packages. The most frequently used commands are update
and install.

Commands:
   update - Retrieve new lists of packages
   upgrade - Perform an upgrade
   install - Install new packages (pkg is libc6 not libc6.deb)
   remove - Remove packages
   source - Download source archives
   dist-upgrade - Distribution upgrade, see apt-get(8)
   dselect-upgrade - Follow dselect selections
   clean - Erase downloaded archive files
   autoclean - Erase old downloaded archive files
   check - Verify that there are no broken dependencies

Options:
  -h  This help text.
  -q  Loggable output - no progress indicator
  -qq No output except for errors
  -d  Download only - do NOT install or unpack archives
  -s  No-act. Perform ordering simulation
  -y  Assume Yes to all queries and do not prompt
  -f  Attempt to continue if the integrity check fails
  -m  Attempt to continue if archives are unlocatable
  -u  Show a list of upgraded packages as well
  -b  Build the source package after fetching it
  -c=? Read this configuration file
```

```
  -o=? Set an arbitary configuration option, eg -o dir::cache=/tmp
See the apt-get(8), sources.list(5) and apt.conf(5) manual
pages for more information and options.
```

It has become a GNU convention to make this type of information accessible with the "−h" and the "−−help" options. Most GNU/Linux users will expect to be able to retrieve basic documentation these ways so if you choose to use different methods, be prepared for the flames and fallout that may result.

2.5.3. Files users will expect

In addition to man pages and command−line help, there are certain files where people will look for documentation, especially in any package containing source code. In a source distribution, most of these files can be stored in the root directory of the source distribution or in a subdirectory of the root called "doc" or "Documentation." Common files in these places include:

README or Readme

> A document containing all the basic installation, compilation, and even basic use instructions that make up the bare minimum information needed to get the program up and running. A README is not your chance to be verbose but should be concise and effective. An ideal README is at least 30 lines long and more no more than 250.

INSTALL or Install

> The INSTALL file should be much shorter than the README file and should quickly and concisely describe how to build and install the program. Usually an INSTALL file simply instructs the user to run "./configure; make; make install" and touches on any unusual options or actions that may be necessary. For most relatively standard install procedures and for most programs, INSTALL files are as short as possible and are rarely over 100 lines.

CHANGELOG, Changelog, ChangeLog, or changelog

> A CHANGELOG is a simple file that every well−managed free software project should include. A CHANGELOG is simple the file that, as its name implies, logs or documents the changes you make to your program. The most simple way to maintain a CHANGELOG is to simply keep a file with the source code for your program and add a section to the top of the CHANGELOG with each release describing what has been changed, fixed, or added to the program. It's a good idea to post the CHANGELOG onto the website as well because it can help people decide whether they want or need to upgrade to a newer version or wait for a more significant improvement.

NEWS

> A NEWS file and a ChangeLog are similar. Unlike a CHANGELOG, a NEWS file is not typically updated with new versions. Whenever new features are added, the developer responsible will make a note in the NEWS file. NEWS files should not have to be changed before a release (they should be kept up to date all along) but it's usually a good idea to check first anyway because often developers just forget to keep them as current as they should.

FAQ

For those of you that don't already know, FAQ stands for Frequently Asked Questions and a FAQ is a collection of exactly that. FAQs are not difficult to make. Simply make a policy that if you are asked a question or see a question on a mailing list two or more times, add the question (and its answer) to your FAQ. FAQs are more optional than the files listed above but they can save your time, increase usability, and decrease headaches on all sides.

2.5.4. Website

It's only indirectly an issue of documentation but a good website is quickly becoming an essential part of any free software project. Your website should provide access to your documentation (in HTML if possible). It should also include a section for news and events around your program and a section that details the process of getting involved with development or testing and make an open invitation. It should also supply links to any mailing lists, similar websites, and provide a direct link to all the available ways of downloading your software.

2.5.5. Other documentation hints

- All your documentation should be in plaintext, or, in cases where it is on your website primarily, in HTML. Everyone can cat a file, everyone has a pager, (almost) everyone can render HTML. *You are welcome to distribute information in PDF, PostScript, RTF, or any number of other widely used formats but this information must also be available in plaintext or HTML or people will be very angry at you.* In my opinion, info falls into this category as well. There is plenty of great GNU documentation that people simply don't read because it only in info. And this *does* make people angry. It's not a question of superior formats; it is a question of accessability and the status quo plays a huge role in this determination.
- It doesn't hurt to distribute any documentation for your program from your website (FAQs etc) with your program. Don't hesitate to throw any of this in the program's tarball. If people don't need it, they will delete it. I can repeat it over and over: *Too much documentation is not a sin.*
- Unless your software is particular to a non−English language (a Japanese language editor for example), please distribute it with English language documentation. If you don't speak English or not not confident in your skills, ask a friend for help. Like it or not, fair or unfair, *English is the language of free software*. However, this does not mean you should limit your documentation to only English. If you speak another language, distribute translations of documentation with your software if you have the time and energy to do so. They will invariably be useful to someone.
- Finally, *please spell−check your documentation.* Misspellings in documentation are bugs. I'm very guilty of committing this error and it's extremely easy to do. If English is not your first language, have a native speaker look over or edit your documentation or web pages. Poor spelling or grammar goes a long way to making your code look unprofessional. In code comments, this type of thing is less important but in man pages and web pages these mistakes are not acceptable.

2.6. Other Presentation Issues

Many of the remaining issues surrounding the creation of a new free software program fall under what most people describe as common sense issues. Its often said that software engineering is 90 percent common sense combined with 10 percent specialized knowledge. Still, they are worth noting briefly in hopes that they may remind a developer of something they may have forgotten.

2.6.1. Package File Names

I agree with ESR when he says that: " 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." There is more info (including lots of examples of what *not* to do in his *Software Release Practices HOWTO* which is included in this HOWTO's bibliography and can be found through the LDP.

2.6.2. Package formats

Package formats may differ depending on the system you are developing for. For windows based software, Zip archives (.zip) usually serve as the package format of choice. If you are developing for GNU/Linux, *BSD, or any UN*X, make sure that your source code is always available in tar'ed and gzip'ed format (.tar.gz). UNIX compress (.Z) has gone out of style and usefulness and faster computers have brought bzip2 (.bz2) into the spot−light as a more effective compression medium. I now make all my releases available in both gzip'ed and bzip2'ed tarballs.

Binary packages should always be distribution specific. If you can build binary packages against a current version of a major distribution, you will only make your users happy. Try to foster relationships with users or developers of large distributions to develop a system for the consistent creation of binary packages. It's often a good idea to provide RedHat RPM's (.rpm), Debian deb's (.deb) and source RPM's SRPM's if possible. Remember: *While these binaries packages are nice, getting the source packaged and released should always be your priority. Your users or fellow developers can and will do the the binary packages for you.*

2.6.3. Version control systems

A version control system can make a lot of these problems of packaging (and a lot of other problems mentioned in this HOWTO) less problematic. If you are using *NIX, CVS is your best bet. I recommend Karl Fogel's book on the subject (and the posted HTML version) wholeheartedly.

CVS or not, you should probably invest some time into learning about a version control system because it provides an automated way of solving many of the problems described by this HOWTO. I am not aware of any free version control systems for Windows or Mac OS but I know that CVS clients exist for both platforms. Websites like SourceForge do a great job as well with a nice, easy−to−use web interface to CVS.

I'd love to devote more space in this HOWTO to CVS because I love it (I even use CVS to keep versions straight on this HOWTO!) but I think it falls outside the scope of this document and already has its own HOWTOs. Most notably is the *CVS Best Practices HOWTO*[CVSBESTPRACTICES] which I've included in the attached bibliography.

2.6.4. Useful tidbits and presentation hints

Other useful hints include:

* *Make sure that your program can always be found in a single location.* Often this means that you have a single directory accessible via FTP or the web where the newest version can be quickly recognized. One effective technique is a provide a symlink called "yourprojectname−latest" that is always pointing to the most recent released or development version of your free software application. Keep in mind that this location will receive many requests for downloads around releases so make sure that the server you choose has adequate bandwidth.
* *Make sure that there is a consistent email address for bug reports.* It's usually a good idea to make this something that is NOT your primary email address like yourprojectname@host or yourprojectname−bugs@host. This way, if you ever decide to hand over maintainership or if your email address changes, you simply need to change where this email address forwards. It also will allow for more than one person to deal with the influx of mail that is created if your project becomes as huge as you hope it will.

---

3. Maintaining a Project: Interacting with Developers

Once you have gotten your project started, you have overcome the most difficult hurdles in the development process of your program. Laying a firm foundation is essential, but the development process itself is equally important and provides just as many opportunities for failure. In the next two sections, I will describe running a project by discussing how to maintain a development effort through interactions with developers and with users.

In releasing your program, your program becomes free software. This transition is more than just a larger user base. By releasing your program as free software, *your* software becomes the *free software community's* software. The direction of your software's development will be reshaped, redirected, and fully determined by your users and, to a larger extent, by other developers in the community.

The major difference between free software development and propriety software development is the developer base. As the leader of a free software project, you need to attract and keep developers in a way that leaders of proprietary software projects simply don't have to worry about. *As the person leading development of a free software project, you must harness the work of fellow developers by making responsible decisions and by responsibly choosing not to make decisions. You have to direct developers without being overbearing or bossy. You need to strive to earn respect and never forget to give it out.*

3.1. Delegating Work

By now, you've hypothetically followed me through the early programming of a piece of software, the creation of a website and system of documentation, and we've gone ahead and (as will be discussed in Section 4.3) released it to the rest of the world. Times passes, and if things go well, people become interested and want to help. The patches begin flowing in.

Like the parent of any child who grows up, it's now time to wince, smile and do most difficult thing in any parents life: It's time to let go.

Delegation is the political way of describing this process of "letting go." It is the process of handing some of the responsibility and power over your project to other responsible and involved developers. It is difficult for anyone who has invested a large deal of time and energy into a project but it essential for the growth of any free software project. One person can only do so much. A free software project is nothing without the involvement of *a group* of developers. A group of developers can only be maintained through respectful and responsible leadership and delegation.

As your project progresses, you will notice people who are putting significant amounts of time and effort into your project. These will be the people submitting the most patches, posting most on the mailing lists, and engaging in long email discussions. It is your responsibility to contact these people and to try and shift some of the power and responsibility of your position as the project's maintainer onto them (if they want it). There are several easy ways you can do this:

In a bit of a disclaimer, delegation need not mean rule by committee. In many cases it does and this has been proven to work. In other cases this has created problems. Managing Projects the Open Source Way argues that "OSS projects do best when one person is the clear leader of a team and makes the big decisions (design changes, release dates, and so on)." I think this often true but would urge developers to consider the ideas that the project leader need not be the project's founder and that these important powers need not all rest with one person but that a release manager may be different than a lead developer. These situations are tricky politically so be careful and make sure it's necessary before you go around empowering people.

3.1.1. How to delegate

You may find that other developers seem even more experienced or knowledgeable than you. Your job as a maintainer does not mean you have to be the best or the brightest. It means you are responsible for showing good judgment and for recognizing which solutions are maintainable and which are not.

Like anything, its easier to watch others delegate than to do it yourself. In a sentence: *Keep an eye out for other qualified developers who show an interest and sustained involvement with your project and try and shift responsibility toward them.* The following ideas might be good places to start or good sources of inspiration:

3.1.1.1. Allow a larger group of people to have write access to your CVS repository and make real efforts toward rule by a committee

Apache is an example of a project that is run by small group of developers who vote on major technical issues and the admission of new members and all have write access to the main source repository. Their process is detailed online.

The Debian Project is an extreme example of rule by committee. At current count, more than 700 developers have full responsibility for aspects of the project. All these developers can upload into the main FTP server, and vote on major issues. Direction for the project is determined by the project's social contract and a constitution. To facilitate this system, there are special teams (i.e. the install team, the Japanese language team) as well as a technical committee and a project leader. The leader's main responsibility is to, "appoint delegates or delegate decisions to the Technical Committee."

While both of these projects operate on a scale that your project will not (at least initially), their example is helpful. Debian's idea of a project leader who can do *nothing* but delegate serves as a caricature of how a project can involve and empower a huge number of developers and grow to a huge size.

3.1.1.2. Publicly appoint someone as the release manager for a specific release

A release manager is usually responsible for coordinating testing, enforcing a code freeze, being responsible for stability and quality control, packaging up the software, and placing it in the appropriate places to be downloaded.

This use of the release manager is a good way to give yourself a break and to shift the responsibility for accepting and rejecting patches onto someone else. It is a good way of very clearly defining a chunk of work on the project as belonging to a certain person and its a great way of giving yourself room to breath.

3.1.1.3. Delegate control of an entire branch

If your project chooses to have branches (as described in Section 3.3), it might be a good idea to appoint someone else to be the the head of a branch. If you like focusing your energy on development releases and the implementation of new features, hand total control over the stable releases to a well−suited developer.

The author of Linux, Linus Torvalds, came out and crowned Alan Cox as "the man for stable kernels." All patches for stable kernels go to Alan and, if Linus were to be taken away from work on Linux for any reason, Alan Cox would be more than suited to fill his role as the acknowledged heir to the Linux maintainership.

3.2. Accepting and Rejecting Patches

This HOWTO has already touched on the fact that as the maintainer of a free software project, one of your primary and most important responsibilities will be accepting and rejecting patches submitted to you by other developers.

3.2.1. Encouraging Good Patching

As the person managing or maintaining the project, you aren't the person who is going to be making a lot of patches. However, it's worth knowing about ESR's section on *Good Patching Practice* in the *Software Release Practices HOWTO*[ESRHOWTO]. I don't agree with ESR's claim that most ugly or undocumented patches are probably worth throwing out at first sight−−this just hasn't been my experience, especially when dealing with bug fixes that often don't come in the form of patches at all. Of course, this doesn't mean that I *like* getting poorly done patches. If you get ugly −e patches, if you get totally undocumented patches, and especially if they are anything more than trivial bug−fixes, it might be worth judging the patch by some of the criteria in ESR's HOWTO and then throwing people the link to the document so they can do it the "right way."

3.2.2. Technical judgment

In *Open Source Development with CVS*, Karl Fogel makes a convincing argument that the most important things to keep in mind when rejecting or accepting patches are:

- A firm knowledge of the scope of your program (that's the "idea" I talked about in Section 2.1);
- The ability to recognize, facilitate, and direct "evolution" of your program so that the program can grow and change and incorporate functionality that was originally unforeseen;
- The necessity to avoid digressions that might expand the scope of the program too much and result and push the project toward an early death under its own weight and unwieldiness.

These are the criteria that you as a project maintainer should take into account each time you receive a patch.

Fogel elaborates on this and states the "the questions to ask yourself when considering whether to implement (or approve) a change are:"

- Will it benefit a significant percentage of the program's user community?
- Does it fit within the program's domain or within a natural, intuitive extension of that domain?

The answers to these questions are never straightforward and its very possible (and even likely) that the person who submitted the patch may feel differently about the answer to these questions than you do. However, if you feel that that the answer to either of those questions is "no," it is your responsibility to reject the change. If you fail to do this, the project will become unwieldy and unmaintainable and many ultimately fail.

3.2.3. Rejecting patches

Rejecting patches is probably the most difficult and sensitive job that the maintainer of any free software project has to face. But sometimes it has to be done. I mentioned earlier (in Section 3 and in Section 3.1) that you need to try and balance your responsibility and power to make what you think are the best technical decisions with the fact that you will lose support from other developers if you seem like you are on a power trip or being overly bossy or possessive of the community's project. I recommend that you keep these three major concepts in mind when rejecting patches (or other changes):

3.2.3.1. Bring it to the community

One of the best ways of justifying a decision to reject a patch and working to not seem like you keep an iron grip on your project is by not making the decision alone at all. It might make sense to turn over larger proposed changes or more difficult decisions to a development mailing list where they can be discussed and debated. There will be some patches (bug fixes, etc.) which will definitely be accepted and some that you feel are so off base that they do not even merit further discussion. It is those that fall into the gray area between these two groups that might merit a quick forward to a mailing list.

I recommend this process wholeheartedly. As the project maintainer you are worried about making the best decision for the project, for the project's users and developers, and for yourself as a responsible project leader. Turning things over to an email list will demonstrate your own responsibility and responsive leadership as it tests and serves the interests of your software's community.

3.2.3.2. Technical issues are not always good justification

Especially toward the beginning of your project's life, you will find that many changes are difficult to implement, introduce new bugs, or have other technical problems. Try to see past these. Especially with added functionality, good ideas do not always come from good programmers. Technical merit is a valid reason to postpone an application of a patch but it is not always a good reason to reject a change outright. Even small changes are worth the effort of working with the developer submitting the patch to iron out bugs and incorporate the change if you think it seems like a good addition to your project. The effort on your part will work to make your project a community project and it will pull a new or less experienced developer into your project and even teach them something that might help them in making their next patch.

3.2.3.3. Common courtesy

It should go without saying but, *above all and in all cases, just be nice.* If someone has an idea and cares about it enough to write some code and submit a patch, they care, they are motivated, and they are already involved. Your goal as the maintainer is make sure they submit again. They may have thrown you a dud this time but next time may be the idea or feature that revolutionizes your project.

It is your responsibility to first justify your choice to not incorporate their change clearly and concisely. Then thank them. Let them know that you a appreciate their help and feel horrible that you can't incorporate their change. Let them know that you look forward to their staying involved and you hope that the next patch or idea meshes better with your project because you appreciate their work and want to see it in your application. If you have ever had a patch rejected after putting a large deal of time, thought, and energy into it, you remember how it feels and it feels bad. Keep this in mind when you have to let someone down. It's never easy but you need to do everything you can to make it as not−unpleasant as possible.

3.3. Stable and Development Branches

The idea of stable and development branches has already been described briefly in Section 2.4 and in Section 3.1.1.3. These allusions attest to some of the ways that multiple branches can affect your software. Branches can let you avoid (to some extent) some of the problems around rejecting patches (as described in Section 3.2) by allowing you to temporarily compromise the stability of your project without affecting those users who need that stability.

The most common way of branching your project is to have one branch that is stable and one that is for development. This is the model followed by the Linux kernel that is described in Section 2.4. In this model, there is *always* one branch that is stable and always one that is in development. Before any new release, the development branch goes into a "feature freeze" as described in Section 3.4.1 where major changes and added features are rejected or put on hold under the development kernel is released as the new stable branch and major development resumes on the development branch. Bug fixes and small changes that are unlikely to have any large negative repercussions are incorporated into the stable branch as well as the development branch.

Linux's model provides an extreme example. On many projects, there is no need to have two versions constantly available. It may make sense to have two versions only near a release. The Debian project has historically made both a stable and an unstable distribution available but has expanded to this to include: stable, unstable, testing, experimental, and (around release time) a frozen distribution that only incorporates bug fixes during the transition from unstable to stable. There are few projects whose size would necessitate a system like Debian's but this use of branches helps demonstrate how they can be used to balance consistent and effective development with the need to make regular and usable releases.

In trying to set up a development tree for yourself, there are several things that might be useful to keep in mind:

Minimize the number of branches

> Debian may be able to make good use of four or five branches but it contains gigabytes of software in over 5000 packages compiled for 5−6 different architectures. For you, two is probably a good ceiling. Too many branches will confuse your users (I can't count how many times I had to describe Debian's system when it only had 2 and sometimes 3 branches!), potential developers and even yourself. Branches can help but they come at a cost so use them very sparingly.

Make sure that all your different branches are explained

> As I mentioned in the preceding paragraph, different branches *will* confuse your users. Do everything you can to avoid this by clearly explaining the different branches in a prominent page on your website and in a README file in the FTP or web directory.

I might also recommend against a mistake that I think Debian has made. The terms "unstable," "testing," and "experimental" are vague and difficult to rank in order of stability (or instability as the case may be). Try explaining to someone that "stable" actually means "ultra stable" and that "unstable" doesn't actually include any unstable software but is really stable software that is untested as a distribution.

If you are going to use branches, especially early on, keep in mind that people are conditioned to understand the terms "stable" and "development" and you probably can't go wrong with this simple and common division of branches.

Make sure all your branches are always available

Like a lot of this document, this should probably should go without saying but experience has taught me that it's not always obvious to people. It's a good idea to physically split up different branches into different directories or directory trees on your FTP or web site. Linux accomplishes this by having kernels in a v2.2 and a v2.3 subdirectory where it is immediately obvious (after you know their version numbering scheme) which directory is for the most recent stable and the current development releases. Debian accomplishes this by naming all their distribution with names (i.e. woody, potato, etc.) and then changing symlinks named "stable," "unstable" and "frozen" to point to which ever distribution (by name) is in whatever stage. Both methods work and there are others. In any case, it is important that different branches are always available, are accessible from consistent locations, and that different branches are clearly distinguished from each other so your users know exactly what they want and where to get it.

3.4. Other Project Management issues

There are more issues surrounding interaction with developers in a free software project that I can not touch on in great detail in a HOWTO of this size and scope. Please don't hesitate to contact me if you see any major omissions.

Other smaller issues that are worth mentioning are:

3.4.1. Freezing

For those projects that choose to adopt a split development model (Section 3.3), freezing is a concept that is worth becoming familiar with.

Freezes come in two major forms. A "feature freeze" is a period when no significant functionality is added to a program. It is a period where established functionality (even skeletons of barely working functionality) can be improved and perfected. It is a period where bugs are fixed. This type of freeze is usually applied some period (a month or two) before a release. It is easy to push a release back as you wait for "one more feature" and a freeze helps to avoid this situation by drawing the much needed line in the sand. It gives developers room they need to get a program ready for release.

The second type of freeze is a "code freeze" which is much more like a released piece of software. Once a piece of software has entered a "code freeze," all changes to the code are discouraged and only changes that fix known bugs are permitted. This type of freeze usually follows a "feature freeze" and directly precedes a release. Most released software is in what could be interpreted as a sort of high level "code freeze."

Even if you never choose to appoint a release manager (Section 3.1.1.2), you will have an easier time justifying the rejection or postponement of patches (Section 3.2) before a release with a publicly stated freeze in effect.

3.5. Forks

I wasn't sure about how I would deal with forking in this document (or if I would deal with forking at all). A fork is when a group of developers takes code from a free software project and actually starts a brand new free software project with it. The most famous example of a fork was between Emacs and XEmacs. Both emacsen are based on an identical code−base but for technical, political, and philosophical reasons, development was split into two projects which now compete with each other.

The short version of the fork section is, *don't do them.* Forks force developers to choose one project to work with, cause nasty political divisions, and redundancy of work. Luckily, usually the threat of the fork is enough to scare the maintainer or maintainers of a project into changing the way they run their project.

In his chapter on "The Open Source Process," Karl Fogel describes how to do a fork if you absolutely must. If you have determined that is absolutely necessary and that the differences between you and the people threatening to fork are absolutely unresolvable, I recommend Fogel's book as a good place to start.

4. Maintaining a Project: Interacting with Users

If you've worked your way up to here, congratulations, you are nearing the end of this document. This final section describes some of the situations in which you, in your capacity as project maintainer, will be interacting with users. It gives some suggestions on how these situations might be handled effectively.

Interacting with users is difficult. In our discussion of interaction with developers, the underlying assumption is that in a free software project, a project maintainer must constantly strive to attract and keep developers who can easily leave at any time.

Users in the free software community are different than developers and are also different than users in the world of proprietary software and they should be treated differently than either group. Some ways in which the groups differ significantly follow:

- The lines between users and developers are blurred in ways that is totally foreign to any proprietary development model. Your users are often your developers and vice versa.
- In the free software world, you are often your users' only choice. Because there is such an emphasis on not replicating the work of others in the free software community and because the element of competition present in the propriety software model is absent (or at least in an extremely different form) in the free software development model, you will probably be the only project that does what you do (or at least the only one that does what you do in the way that you do it). This means your responsiveness to your users is even more important than in the proprietary software world.
- In an almost paradoxical situation, free software projects have less immediate or dire consequences for ignoring their users altogether. It is also often easier to do. Because you don't usually need to compete with another product, chances are good that you will not be scrambling to gain the features of your competitor's newest program. This means that your development process will have to be directed either internally, by a commitment to your users, or through both.

Trying to tackle this unique situation can only be done indirectly. Developers and maintainers need to listen to users and to try and be as responsive as possible. A solid knowledge of the situation recounted above is any free software developer's best tool for shifting his development or leadership style to fit the unique process of free software project management. This chapters will try and introduce some of the more difficult or important points in any projects interactions with users and give some hints on how to tackle these.

4.1. Testing and Testers

In addition to your users being your developers, they are also (and perhaps more commonly) your testers. Before I get flamed, I should rephrase my sentence: *some of your users* (those who explicitly volunteer) are your testers.

It is important that this distinction be made early on because not all of your users want to be testers. Many users want to use stable software and don't care if they don't have the newest, greatest software with the latest, greatest features. These users except a stable, tested piece of software without major or obvious bugs and will be angry if they find themselves testing. This is yet another way in which a split development model (as mentioned in Section 3.3) might come in handy.

"Managing Projects the Open Source Way" describes what a good test should look for:

Boundary conditions

Maximum buffer lengths, data conversions, upper/lower boundary limits, and so on.

Inappropriate behavior

Its a good idea to find out what a program will do if a user hands it a value it isn't expecting, hits the wrong button, etc. Ask yourself a bunch of "what if" questions and think of anything that *might* fail or *might* go wrong and find out what your program would do in those cases.

Graceful failure

The answer to a number of the "what if" questions above is probably "failure" which is often the only answer. Now make sure that it happens nicely. Make sure that when it crashes, there is some indication of why it crashed or failed so that the user or developer understands whats going on.

Standards conformance

If possible, make sure your programs conforms to standards. If it's interactive, don't be too creative with interfaces. If it is non−interactive, make sure it communicates over appropriate and established channels with other programs and with the rest of the system.

4.1.1. Automated testing

For many programs, many common mistakes can be caught by automated means. Automated tests tend to be pretty good at catching errors that you've run into several times before or the things you just forget. They are not very good at finding errors, even major ones, that are totally unforeseen.

CVS comes with a Bourne shell script called sanity.sh that is worth looking at. Debian uses a program called lintian that checks Debian packages for all of the most common errors. While use of these scripts may not be helpful, there is a host of other sanity checking software on the net that may be applicable (feel free to email me any recommendations). None of these will create a bug−free release but they will avoid at least some major oversights. Finally, if your programs become a long term endeavor, you will find that there are certain errors that you tend to make over and over. Start a collection of scripts that check for these errors to help keep them out of future releases.

4.1.2. Testing by testers

For any program that depends on user interactivity, many bugs will only be uncovered through testing by users actually clicking the keys and pressing the mouse buttons. For this you need testers and as many as possible.

The most difficult part of testing is finding testers. It's usually a good tactic to post a message to a relevant mailing list or news group announcing a specific proposed release date and outlining the functionality of your program. If you put some time into the announcement, you are sure to get a few responses.

The second most difficult part of testing is *keeping* your testers and keeping them actively involved in the testing process. Fortunately, there are some tried and true tactics that can applied toward this end:

Make things simple for your testers

Your testers are doing you a favor so make it as easy as possible for them. This means that you should be careful to package your software in a way that is easy to find, unpack, install, and uninstall. This also means you should explain what you are looking for to each tester and make the means for reporting bugs simple and well established. The key is to provide as much structure as possible to make your testers' jobs easy and to maintain as much flexibility as possible for those that want to do things a little differently.

Be responsive to your testers

When your testers submit bugs, respond to them and respond quickly. Even if you are only responding to tell them that the bug has already been fixed, quick and consistent responses make them feel like their work is heard, important, and appreciated.

Thank your testers

Thank them personally each time they send you patch. Thank them publicly in the documentation and the about section of your program. You appreciate your testers and your program would not be possible without their help. Make sure they know it. Publicly, pat them on the back to make sure the rest of the world knows it too. It will be appreciated more than you expected.

4.2. Setting up Support Infrastructure

While testing is important, the large part of your interactions and responsibility to your users falls under the category of support. The best way to make sure your users are adequately supported in using your program is to set up a good infrastructure for this purpose so that your developers and users help each other and less of the burden falls on you. This way, people will also get quicker and better responses to their questions. This infrastructure comes in several major forms:

4.2.1. Documentation

It should not come as any surprise that the key element to any support infrastructure is good documentation. This topic was largely covered in Section 2.5 and will not be repeated here.

4.2.2. Mailing lists

Aside from documentation, effective mailing lists will be your greatest tool in providing user support. Running a mailing list well is more complicated than installing mailing list software onto a machine.

4.2.2.1. Separate lists

A good idea is too separate your user and development mailing lists (perhaps into project−user@host and project−devel@host) and enforce the division. If people post a development question onto −user, politely ask them to repost it onto −devel and vise versa. Subscribe yourself to both groups and encourage all primarily developers to do the same.

This system provides so that no one person is stuck doing all of the support work and works so that users learn more about the program, they can help newer users with their questions.

4.2.2.2. Choose mailing list software well

Please don't make the selection of mailing list software impulsively. Please consider easy accessibility by users without a lot of technical experience so you want to be as easy as possible. Web accessibility to an archive of the list is also important.

The two biggest free software mailing list programs are majordomo and GNU Mailman. A long time advocate of majordomo, I would now recommend any project choose GNU Mailman. It fulfills the criteria listed above and makes it easier. It provides a good mailing list program for a free software project maintainer as opposed to a good mailing list application for a mailing list administrator.

There are other things you want to take into consideration in setting up your list. If it is possible to gate your mailing lists to Usenet and provide it in digest form as well as making them accessible on the web, you will please some users and work to make the support infrastructure slightly more accessible.

4.2.3. Other support ideas

A mailing list and accessible documentation are far from all you can do to set up good user support infrastructure. Be creative. If you stumble across something that works well, email me and I'll include it here.

4.2.3.1. Make your self accessible

You can not list too few methods to reach you. If you hang out in an IRC channel, don't hesitate to list it in your projects documentation. List email and snailmail addresses, and ways to reach you via ICQ, AIM, or Jabber if they apply.

4.2.3.2. Bug management software

For many large software projects, use of bug management software is essential to keep track of which bugs have been fixed, which bugs have not been fixed, and which bugs are being fixed by which people. Debian uses the Debian Bug Tracking System (BTS) although it may not be best choice for every project (it seems to currently be buckling under its own weight) As well as a damn good web browser, the Mozilla project has spawned a sub−project resulting in a bug tracking system called bugzilla which has become extremely possible and which I like a lot.

These systems (and others like them) can be unwieldy so developers should be careful to not spend more time on the bug tracking system than on the bugs or the projects themselves. If a project continues to grow, use of a bug tracking system can provide an easy standard avenue for users and testers to report bugs and for developers and maintainers to fix them and track them in an orderly fashion.

4.3. Releasing Your Program

As mentioned earlier in the HOWTO, the first rule of releasing is, *release something useful.* Non−working or not−useful software will not attract anyone to your project. People will be turned off of your project and will be likely to simply gloss over it next time they see a new version announced. Half−working software, if useful, will intrigue people, whet their appetites for versions to come, and encourage them to join the development process.

4.3.1. When to release

Making the decision to release your software for the first time is an incredibly important and incredibly stressful decision. But it needs to done. My advice is to try and make something that is complete enough to be usable and incomplete enough to allow for flexibility and room for imagination by your future developers. It's not an easy decision. Ask for help on a local Linux User Group mailing list or from a group of developer friends.

One tactic is to first do an "alpha" or "beta" release as described below in Section 4.3.3. However, most of the guidelines described above still apply.

When you feel in your gut that it is time and you feel you've weighed the situation well several times, cross your fingers and take the plunge.

After you've released for the first time, knowing when to release becomes less stressful, but just as difficult to gauge. I like the criteria offered by Robert Krawitz in his article, "Free Software Project Management" for maintaining a good release cycle. He recommends that you ask yourself, "does this release..."

- Contain sufficient new functionality or bug fixes to be worth the effort.
- Be spaced sufficiently far apart to allow the user time to work with the latest release.
- Be sufficiently functional so that the user can get work done (quality).

If the answer is yes to all of these questions, its probably time for a release. If in doubt, remember that asking for advice can't hurt.

4.3.2. How to release

If you've followed the guidelines described in this HOWTO up until this point, the mechanics of doing a release are going to be the easy part of releasing. If you have set up consistent distribution locations and the other infrastructure described in the preceding sections, releasing should be as simple as building the package, checking it once over, and uploading it into the appropriate place and then making your website reflect the change.

4.3.3. Alpha, beta, and development releases

When contemplating releases, it worth considering the fact that not every release needs to be a full numbered release. Software users are accustomed to pre−releases but you must be careful to label these releases accurately or they will cause more problems then they are worth.

The observation is often made that many free software developers seem to be confused about the release cycle. "Managing Projects the Open Source Way" suggests that you memorize the phrase, "Alpha is not Beta. Beta is not Release" and I'd agree that tis is a probably a good idea.

alpha releases

Alpha software is feature−complete but sometimes only partially functional.

Alpha releases are expected to be unstable, perhaps a little unsafe, but definitely usable. They *can* have known bugs and kinks that have yet to be worked out. Before releasing an alpha, be sure to keep in mind that *alpha releases are still releases* and people are not going to be expecting a nightly build from the CVS source. An alpha should work and have minimal testing and bug fixing already finished.

beta releases

Beta software is feature−complete and functional, but is in the testing cycle and still has a few bugs left to be ironed out.

Beta releases are general expected to be usable and slightly unstable, although definitely *not unsafe.* Beta releases usually preclude a full release by under a month. They can contain small known bugs but no major ones. All major functionality should be fully implemented although the exact mechanics can still be worked out. Beta releases are great tool to whet the appetites of potential users by giving them a very realistic view of where your project is going to be in the very near future and can help keep interest by giving people *something.*

development releases

"Development release" is much a more vague term than "alpha" or "beta". I usually choose to reserve the term for discussion of a development branch although there are other ways to use the term. So many in fact, that I feel the term has been cheapened. The popular window manager Enlightenment has released *nothing but* development releases. Most often, the term is used to describe releases that are not even alpha or beta and if I were to release a pre−alpha version of a piece of software in order to keep interest in my project alive, this is probably how I would have to label it.

4.4. Announcing Your Project

Well, you've done it. You've (at least for the purposes of this HOWTO) designed, built, and released your free software project. All that is left is for you to tell the world so they know to come and try it out and hopefully jump on board with development. If everything is in order as described above, this will be a quick and painless process. A quick announcement is all that it takes to put yourself on the free software community's radar screen.

4.4.1. Mailing lists and Usenet

Announce your software on Usenet's comp.os.linux.announce. If you only announce your software in two places, have it be c.o.l.a and freshmeat.

However, email is still the way that most people on the Internet get their information. Its a good idea to send a message announcing your program to any relevant mailing list you know of and any other relevant Usenet discussion groups.

Karl Fogel recommends that use you simple subject describing the fact that the message is an announcement, the name of the program, the version, and a half−line long description of its functionality. This way, any interested user or developer will be immediately attracted to your announcement. Fogel's example looks like:

```
Subject: ANN: aub 1.0, a program to assemble Usenet binaries
```

The rest of the email should describe the programs functionality quickly and concisely in no more than two paragraphs and should provide links to the projects webpage and direct links to downloads for those that want to try it right away. This form will work for both Usenet and mailing list posts.

You should repeat this announcement process consistently in the same locations for each subsequent release.

4.4.2. freshmeat.net

Mentioned earlier in Section 2.1.2.1, in today's free software community, announcements of your project on freshmeat are almost more important than announcements on mailing lists.

Visit the freshmeat.net website or their submit project page to post your project onto their site and into their database. In addition to a large website, freshmeat provides a daily newsletter that highlights all the days releases and reaches a huge audience (I personally skim it every night for any interesting new releases).

4.4.3. Project Mailing List

If you've gone ahead and created mailing lists for your project, you should always announce new versions on these lists. I've found that for many projects, users request a very low−volume announce only mailing list to be notified when new versions are released. freshmeat.net now allows users to subscribe to a particular project so they receive emails every time a new version is announced through their system. It's free and it can stand in for an announce−only mailing list. In my opinion, it can't hurt.

Bibliography

Printed Books

Karl Fogel, *Open Source Development with CVS*, Coriolois Open Press, 1999, 1−57610−490−7.

Fogel's "guide to using CVS in the free software world" is much more than its subtitle. In the publisher's own words: "*Open Source Development with CVS* is one of the first books available that teaches you development and implementation of Open Source software." It also includes the best reference and tutorial to CVS I have ever seen. It is the book that was *so good* that it prompted me to write this HOWTO because I thought the role it tried to serve was so important and useful. Please check it or buy it if you can and are seriously interested in running a free software project.

Lawrence Lessig, *Code and Other Laws of Cyberspace*, Basic Books, 2000, 0−465−03913−8.

While it only briefly talks about free software (and does it by tiptoeing around the free software/open source issue with the spineless use of the term "open code" that only a lawyer could coin), Lessig's book is brilliant. Written by a lawyer, it talks about how regulation on the Internet is not done with law, but with the code itself and how the nature of the code will determine the nature of future freedoms. In addition to being a quick and enjoyable read, it gives some cool history and describes how we *need* free software in a way more powerfully than anything I've read outside of RMS's "Right to Read."

Eric Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*, O'Reilly, 1999, 1−56592−724−9.

Although I have to honestly say that I am not the ESR fan that I used to be, this book proved invaluable in getting me where I am today. The essay that gives the book its title does a good job of sketching the free software process and does an an amazing job of making an argument for free software/open source development as a road to better software. The rest of the book has other of ESR's articles, which for the most part are posted on his website. Still, it's nice thing to own in hard copy and something that every free software/open source hacker should read.

Web−Accessible Resources

George N Dafermos, *Management and Virtual Decentralized Networks: The Linux Project*.

Since the paper includes its own abstract, I thought I would include it here verbatim:

> This paper examines the latest of paradigms − the Virtual Network(ed) Organisation − and whether geographically dispersed knowledge workers can virtually collaborate for a project under no central planning. Co−ordination, management and the role of knowledge arise as the central areas of focus. The Linux Project and its development model are selected as a case of analysis and the critical success factors of this organisational design are identified.

> The study proceeds to the formulation of a framework that can be applied to all kinds of virtual decentralised work and concludes that value creation is maximized when there is intense interaction and uninhibited sharing of information between the organisation and the surrounding community. Therefore, the potential success or failure of this organisational paradigm depends on the degree of dedication and involvement by the surrounding community.

This paper was referred to me in my capacity as author of this HOWTO and I was very impressed. It's written by a graduate student in management and I think it succeeds at evaluating the Linux project as an example of a new paradigm in management−−one that *you* will be be placing yourself at the center of in your capacity as maintainer of a free software project.

As a developer trying to control an application and guide it to success in the free software world, I'm not sure how useful Dafermos's argument is. It does however, provide a theoretical justification for my HOWTO−−free software project management *is* a different creature than proprietary software project management. If you are interested in the conceptual and theoretical ways that free software project management differs from other types of management, this is a great paper to read. If this paper answers questions of "how?", Dafermos answers the (more difficult to defend) questions of "why?" and does a very good job.

Richard Gabriel, *The Rise of "Worse is Better"*.

A well written article although I think the title may have confused as many people as the rest of the essay helped. It offers a good description of how to design programs that will succeed and stay maintainable as they grow.

Montey Manley, *Managing Projects the Open Source Way*, Linux Programming, Oct 31, 2000.

In one of the better articles on the subject that I've read, Monty sums up some of the major points I touch on including: starting a project, testing, documentation, organizing a team and leadership, and several other topics. While more opinionated that I try to be, I think its an important article that I found very helpful in writing this HOWTO. I've tried to cite him in the places where I borrowed from him most.

I have problems much of this piece and I recommend you read [KRAWITZ] at the same time you read Monty's article for a good critique.

Eric Steven Raymond, *Software Release Practice HOWTO*.

At first glance, ESR's release practice HOWTO seems to share a lot of terrain with this document. Upon closer examination, the differences become apparent but they are closely related. His document, read in conjunction with mine, will give a reader a good picture of how to go about managing a project. ESR's HOWTO goes into a bit more detail on how to write and what languages to write in. He tends to give more specific instructions and checklists ("name this file this, not this") while this HOWTO speaks more conceptually. There are several sections that are extremely similar. It's also *much* shorter.

My favorite quote from his HOWTO is: ""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." Oh really? Perhaps I just do a poor job.

Vivek Venugopalan, *CVS Best Practices*.

Venugopalan provides one of the best essays on effective use of CVS that I've come across. It is written for people who already have a good knowledge of CVS. In the chapter on branching, he describes when and how to branch but gives no information on what CVS commands you should use to do this. This is fine (technical CVS HOWTO have been written) but CVS newbies will want to spend some time with Fogel's reference before they will find this one very useful.

Venugopalan creates checklists of things to do before, after, and around releases. It's definitely worth a read through as most of his ideas will save tons of developer head aches over any longer period of time.

Advogato Articles

Stephen Hindle, *'Best Practices' for Open Source?*, Advogato, March 21, 2001.

Touching mostly on programming practice (as most articles on the subject usually do), the article talks a little about project management ("Use it!") and a bit about communication within a free software project.

Bram Cohen, *http://www.advogato.org/article/258.htmlHow to Write Maintainable Code*, Advogato, March 15, 2001.

This article touches upon the "writing maintainable code" discussion that I try hard to avoid in my HOWTO. It's one of the better (and most diplomatic) articles on the subject that I've found.

Robert Krawitz, *Free Source Project Management*, Advogato, November 4, 2000.

This article made me happy because it challenged many of the problems that I had with Monty's article on LinuxProgramming. The author argues that Monty calls simply for the application of old (proprietary software) project management techniques in free software projects instead of working to come up with something new. I found his article to be extremely well thought out and I think it's an essential read for any free software project manager.

Lalo Martins, *Ask the Advogatos: why do Free Software projects fail?*, Advogato, July 20, 2000.

While the article is little more than a question, reading the answers to this question offered by Advogato's readers can help. In a lot of ways, this HOWTO acts as my answer to the questions posed in this article but there are others, many of which might take issue with whats is in this HOWTO. It's worth checking out.

David Burley, *In−Roads to Free Software Development*, Advogato, June 14, 2000.

This document was written as a response to another Advogato article. Although not about running a project, this describes some of the ways that you can get started with free software development without starting a project. I think this is an important article. If you are interested in becoming involved with free software, this article showcases some of the ways that you can do this without actually starting a project (something that I hope this HOWTO has demonstrated is not to be taken lightly).

Jacob Moorman, *Importance of Non−Developer Supporters in Free Software*, , Advogato, April 16, 2000.

Moorman's is a short article but it brings up some good points. The comment reminding developers to thank their testers and end−users is invaluable and oft−forgotten.

Leslie Orchard, *On Naming an Open Source Project*, Advogato, April 12, 2000.

I didn't even have a section on project naming in this HOWTO (See Section 2.2) until Leslie Orchard's article reminded me of it. Thanks to Leslie for writing this article!

David Allen, *Version Numbering Madness*, Advogato, February 28, 2000.

In this article, David Allen challenges the whole "Major.Minor.Patch" version numbering scheme. Its good to read this as you read Section 2.4. I liked the article and it describes some of the projects that I bring up in my discussion of version numbering.

A. GNU Free Documentation License

A.1. 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non−commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

A.2. 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front−matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front−Cover Texts or Back−Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine−readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard−conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine−generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A.3. 2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or non−commercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

A.4. 3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front−Cover Texts on the front cover, and Back−Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine−readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly−accessible computer−network location containing a complete Transparent copy of the Document, free of added material, which the general network−using public has access to download anonymously at no charge using public−standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

A.5. 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- **A.** Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- **B.** List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- **C.** State on the Title Page the name of the publisher of the Modified Version, as the publisher.
- **D.** Preserve all the copyright notices of the Document.
- **E.** Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- **F.** Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- **G.** Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- **H.** Include an unaltered copy of this License.
- **I.** Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- **J.** Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

- **L.** Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

- **M.** Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

- **N.** Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front−matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties−−for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front−Cover Text, and a passage of up to 25 words as a Back−Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front−Cover Text and one of Back−Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version .

A.6. 5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

A.7. 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and dispbibute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

A.8. 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self–contained works thus compiled with the Document , on account of their being thus compiled, if they are not themselves derivative works of the Document. If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

A.9. 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

A.10. 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

A.11. 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.