www.freesoftwaremagazine.com

Issue 8 - October, 2005

# Free Software MAGAZINE

## Liberating Software

**Games in captivity** (Matt Barton)
Liberation, emulation, and abandonware

**Emulation** (Matt Barton)
Bridges over troubled waters

**Initialization sequence in GNU/Linux** (Steven Goodwin)
The process of booting your PC, from power to prompt

**Interview with Roberto Vacca** (Gianluca Pignalberi)
Gianluca interviews Roberto Vacca, the Italian engineer, who is known as a futurologist because of his sharp forecasts and provisional models

**Towards a free matter economy (Part 2)** (Terry Hancock)
The passing of the shade tree mechanic

**Replacing proprietary anti-virus software** (Robin Monks)
Using ClamWin Free Antivirus to replace proprietary anti-virus software

**Free software and Latin America** (David Sugar)
The challenge is often political rather than technical

**Working together and sharing code with TLA** (Carlo Contavalli)
TLA as your Version Control System

COVER BY
WWW.QREATE.NET

ISSN 1746-8752

9 771746 875009 08

E-LOCA-TRONICA 2005
STICK TO WHAT YOU DON'T KNOW

# HTTP://WWW.LOCARECORDS.COM

# INFO@LOCARECORDS.COM













LOCA RECORDS, PO BOX 233,
BRIGHTON BN2 3WX, UK

# Free Software MAGAZINE

# Contents

Issue 8, October 2005

# What do so many online communities have in common?

# Drupal

STANDARDS COMPLIANT
ACCESSIBLE
MODULAR
STABLE
FREE

Visit **drupal.org** to find out why more grassroots campaigns, interest groups and corporations are switching to Drupal to manage their community and establish their online presence.

# Let's take care of our memory

It was late at night in Sydney. I was at John Paul's house—the man behind MySource (http://mysource.squiz.net/). We hadn't seen each other for years, and we had spent the whole day helping his parents move house, so we did what old friends do: we talked about anything and everything. The conversation somehow turned to neural damage and freak accidents (our backs must have hurt).

I remembered something someone (Dave Guard?) told me years earlier, and decided to contribute to the never-ending spread of useless information (typical of the human race): "Well, I heard about this guy who got a chunk of wood shot through his skull, and he was sort-of fine afterwards. The chunk of wood went right through his brain, and was stuck in his head... Can you imagine the doctors in the ER?".

We laughed. Then, he asked "I've heard that story too! Didn't it happen to someone from England?"

In the pre-internet world, the conversation would have ended there, in uncertainty and doubt. If we really wanted to know, we could have called up some neurosurgeons in the morning and asked them to give us some medical references, but I don't think we would have bothered.

However in the post-internet world, we looked at each other in silence for a couple of seconds thinking of how we could find out more, and we both came up with one meaningful word: "Google".

Finding information on Linux in Google is quite simple. Finding out about people who have had chunks of wood lodged in their heads, however, is tricky to say the least. In the end, we managed to find a few "reliable" bits of info (and discovered that we had both been wrong), but it was a big struggle.

This episode raised a question in my mind: has the web effectively become the world's memory? And has Google become the way of fetching anything from this memory?

If the answer to these questions is "yes", then there are some issues that should be considered.

The first one is: what about things that happened before the web existed? In general, particularly after blogging began, current affairs events are thoroughly talked about and discussed. But anything that happened in prehistory (in this case, anything that occurred before 1999) is published on the web as reported, second hand information. If the web is the new memory of the world, it is a very selective memory and only recent events are remembered vividly. The rest is old news and isn't easily accessible.

The web can also be poisoned (contents-wise) quite easily. Anyone can go about publishing anything on web, and then give it relevance artificially by using various tricks.

Blatantly false information is mixed with more accurate facts and no one can decide what's accurate and what's not. Ironically, human brains create "false memories" as well (ask Google if you don't believe me!).

What about the information that very few, perhaps disadvantaged, people care about? This is a more stringent issue. While looking for information about Linux is quite easy, if you look for the best coffee shop in Nicaragua, you might be out of luck. You don't have to be that obscure to find holes in the web-memory. I have tried searching on animal health, for example, and it was more tedious than you can imagine...

These are quite common questions and I could have raised even more intriguing questions. The internet (as well as search engines), however, somehow seems to be evolving quite well in order to deal with all of those problems.

The one problem that I consider "scary" is: what if the interface to this enormous wealth of information, Google, disappeared? Or what if it became a "pay-per-use" service? Or what if it became less "ethical" and more selective about the results it gives?

In this case, this enormous wealth of information, the web, the world's memory, could become inaccessible, and therefore completely useless. Though just now, this outcome seems very unlikely. However, companies do change their policies—especially when they are short of cash...

They sometimes collapse, or are bought by another company. If Google's stock price became $1 tomorrow, the obvious alternative to it might be... Microsoft's MSN. How encouraging.

If it's true that the web is becoming the world's memory, regardless of its problems, then I feel that we, the creators and owners of this immense wealth of information, ought to have a free (free as in freedom) way of accessing it. A way that doesn't depend on the health or ethical decisions of a particular company (Google, Microsoft, or whatever). A way that doesn't require a rack of 400 servers (and growing) or immense amounts of bandwidth in order to index what's there and process it.

Unfortunately, nobody has come up with a solution yet. However, any ideas are most definitely welcome.





**Free Software Magazine** is a magazine by **The Open Company Partners Inc.**
90 Main St. Road Town, Tortola BVI

**EDITOR IN CHIEF**
Tony Mobily (t.mobily@)

**TECHNICAL EDITORS**
Clare James (c.james@)
Pancrazio De Mauro (p.demauro@)

**EDITORS**
Anna Dymitr Hawkes (a.dymitrhawkes@)
Dave Guard (d.guard@)

**ADVERTISING AND SALES**
Stephen Wetzel (s.wetzel@)

**TECHS**
Gianluca Pignalberi (LaTeX class and magazine generation) (g.pignalberi@)
Gian Maria Ricci (RTF to XML converter using VBA) (gm.ricci@)

**GRAPHIC DESIGN**
Alan Sprecacenere (Web, cover and advertising design) (a.sprecacenere@)
Tony Mobily, Gianluca Pignalberi, Alan Sprecacenere (Magazine design)

**CONTRIBUTORS**
Matt Barton, Martin C Brown, Carlo Contavalli, Steven Goodwin, Terry Hancock, Aaron E. Klemm, Robin Monks, Gianluca Pignalberi, David Sugar.

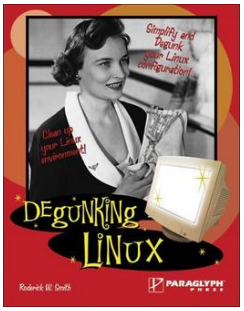
**THIS PROJECT EXISTS THANKS TO**
Donald E. Knuth, Leslie Lamport, People at TeX Users Group TUG (http://www.tug.org)

Every listed person is contactable by email. Please just add freesoftwaremagazine.com to the person's username in parentheses.

*For copyright information about the contents of Free Software Magazine, please see the section "Copyright information" at the end of each article.*

# Degunking Linux *by Roderick W Smith*

Martin C Brown




Over the course of a typical computer's lifetime you will probably create all sorts of files, temporarily install software and generate lots of information and data that you don't really want to keep. Unfortunately, computers tend to have a terrible habit of keeping these files and information about. In Degunking Linux by Roderick W Smith you'll find hints on how to clean and, as the title suggests, degunk your Linux installation to help free up disk space, CPU time and help optimize your machine. You'd be amazed how much of a difference degunking your machine can make. Not only do you get more disk space but you also have the potential to speed up your machine and make it more stable.

There's a 12-step program of degunking as well as a list of degunking methods that you can perform in 15 minutes

## The contents

Degunking Linux is split into four main sections and is spread over 12 chapters. The first section looks at basic degunking techniques such as sorting your files, settings and other information. There are lots of good tips in this section, including theories behind some of the techniques such as deleting unused accounts, identifying unused files and sorting out your hardware and drivers. This is mostly an overview section covering the fundamental techniques and pointers to other parts of the book for more information.

The secons section gets into the basics of sorting user data; your old files, desktop environments and settings for applications that you no longer use. The aim here is to help identify and then delete settings, options and files you don't use. It includes "dot" files from your home directory, removing old desktop patterns and pictures and so on. The guide covers both Gnome and KDE environments and goes as deep as configuring browser settings like caches and JavaScript/Java files and settings.

Removing system components—including managing software and packages, deleting accounts, software performance and process management—feature in the third section. This is best of the sections as it provides an overall guide that will help improve performance for all users. Package management, for example, will help to remove drivers and applications that you have installed on your machine but which you don't use. Even with a "minimal" installation of most Linux distributions you'll often end up with tools and utilities you don't want. Others might have been superseded by other applications; for example Gentoo installs nano as the default text editor, but some will prefer vim or emacs and will no longer need nano.

The final section goes much deeper into your system and network to not only degunk existing systems but also to try and prevent your system getting gunked up with information you don't need. For example, there are step by step guides on how to set up, identify and configure the drivers required by your system and to keep it up-to-date, removing the older drivers when they are no longer required. There are also many tips on protecting yourself through the use of virus checking, web proxies (to help remove web sites you don't need) and filtering your email for SPAM.

## Who's this book for?

I think it is fair to say that the book was targeted at desktop users and those who use Linux regularly as their main OS. The book approaches many of the tasks from the perspective of someone who probably isn't that aware of what goes on behind the scenes and what sort of an effect this could have their machine.

However, even with this approach there's lots that can be used by administrators to help degunk their servers and there's no reason why the information and tips given can't be employed by administrators to be applied to their user desktops and systems, or even to form the basis of a guide for users.

## Pros

The best feature of the book is actually a little section at the start of the title, beginning from the inside front cover, which provides some quick, time-based tips for degunking. There's a 12-step program of degunking as well as a list of degunking methods that you can perform in increments of 15 minutes, 30 minutes, one hour, three hours and half a day. Even following the tips in the 15-minute section will provide some benefit. So if you've got "free" time, there's a section with steps you can take to help degunk your machine. In each case you get a simple description and the page number where the relevant details are covered.

It's nice to see that it's not just cleaning and deleting being covered by the book, but also complete techniques for managing your machine, such as backing up critical files and how leaving some of the elements (user accounts, unknown software) on your computer can be a security risk.

There are many places where Roderick covers a particular element that doesn't strike you as an obvious source of problems, but when given a little thought highlight a potential area of issue. For example, he covers the organization of mailboxes and aliases that might be enabled—or in some cases required—on your machine. This is just one example of how detailed and exhaustive Roderick has covered the material.

Overall there's good coverage here for both repairing and cleaning your system and for all the preventative maintenance required to keep your machine comparatively gunk free.

## Cons

There are a couple of places where I would have liked a little bit depth. For example, although viruses are mentioned there isn't any coverage of a virus tool, like ClamAV, which I would have considered a required tool. I would also like to have seen coverage of a CD distribution like Knoppix that can make some of the procedures covered (like the Virus checking) much easier.

However, these are only minor niggles and don't detract from the excellent coverage and material in the rest of the book.

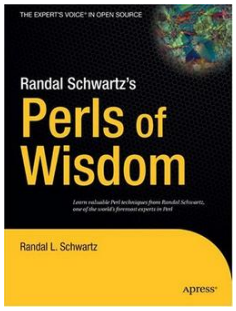
## In short

| | |
|---|---|
| Title | Degunking Linux |
| Author | Roderick W Smith |
| Publisher | Paraglyph Press |
| ISBN | 1-933097-04-3 |
| Year | 2005 |
| Pages | 332 |
| CD included | No |
| Mark | 9 |

## Copyright information

# Randal Schwartz's Perls of Wisdom *by Randal L Schwartz*

Martin C Brown




Ask for some key figures in the world of Perl and it won't be long before the name Randal L Schwartz appears. Randal has, at one time or another, been a trainer of Perl, the Pumpking (responsible for managing the development of Perl), as well as a prolific writer and speaker on Perl techniques and materials. In Perls of Wisdom (Apress) he gathers together many of his talks and articles into a single book, expanding, correcting and extending them as necessary.

> Randal knows Perl so well there are some absolute gems here on how to solve specific problems using techniques and algorithms that are far from obvious, but just as effective

## The contents

There are only five chapters in this book (spread over 343 pages), but each chapter is made up of a number of specific examples. In each case, the example is based on an article, column or email/Usenet post that Randal has used to cover a specific issue. You get coverage of the problem, a detailed example (including line-numbered source code) and a step-by-step guide of the solution and how it works. These are not snippets of potential code. Nor are they just an explanation of the code itself, you get full details on the methodologies used and how they can be adapted or changed to best suit your own needs and applications.

> This is the best collection of real-world Perl knowledge available in a single book

Because each example is an article that has previously been published, Randal also takes the opportunity to provide some commentary on how he might have done things differently and even provides a few corrections and updates in places. Having been previously published, he's also had the opportunity to debug the examples and the code so you end up with some very clear and coherent samples.

The content is varied. The five chapter headings: Advanced Perl Techniques, Test Searching and Editing, HTML and XML Processing, CGI Programming and Webmaster Tools, don't remotely give the content the credit it deserves. There are some amazing samples here, from the basics of object programming right up to load balancing scripts and web cookie tools.

There are examples covered are so good that I think they should be required reading for any Perl programmer. The aforementioned object introduction for example would go a long way to helping many Perl programmers make better use of the class system built into Perl. "Discovering Incomprehensible Documentation" should be an application fitted to the output of every man page generated by Perl programmers, although I sadly doubt it will address all of the issues we regularly experience when trying to understand the documentation included with certain modules I wont mention.

Other examples provide more useful tools that we should all be exploiting. There's an automatic meta-indexing tool (for web pages) that could be the key part of many website searching tools and the calendar sample could the form the basis of an excellent web-based diary application.

All of these examples are provided with such finesse and style that it can be difficult to appreciate that you are actually reading a guide to writing code for one of the most popular, easy to use and on so many occasions confusing programming languages available.

## Who's this book for?

Perl programmers, pure and simple. If you program in Perl, read this book. I don't care what you think you already know, read it anyway. Even if you could have guessed at Randal's solution, his prose and relaxed style is incredible easy and enjoyable to read.

If I were to recommend the book to any other group of people it would be writers, just to give them an example of how to write examples and code in a way that makes it useful and easy to understand.

## Pros

Quite possibly this is the best condensation of real-world Perl knowledge available in a single book. It far exceeds the information and examples provided in any cookbook, or even the titles

from O'Reilly in terms of actually addressing common user needs and problems. Because the book is based, mostly, on real reader requests and problems that Randal himself has faced it goes much further than the usual example-led approaches.

Furthermore, because Randal knows Perl so well, there are some absolute gems here on how to solve specific problems using techniques and algorithms that are far from obvious, but just as effective. The book not only answers your queries, it also shows you elements of Perl that you'd forgotten, or simply didn't realize existed, and in such a way that it's easy to see how you could adapt it.

> If you program in Perl, read this book. It doesn't matter what you think you already know, read it anyway

## Cons

The content ends far too quickly. I eagerly await "More Perls of Wisdom".

## In short

| | |
|---|---|
| Title | Perls of Wisdom |
| Author | Randal L Schwartz |
| Publisher | Apress |
| ISBN | 1-59059-323-5 |
| Year | 2005 |
| Pages | 348 |
| CD included | No |
| Mark | 10 |

## Copyright information

# Interview with Roberto Vacca

## Gianluca interviews Roberto Vacca, the Italian engineer, who is known as a "futurologist" because of his sharp forecasts and provisional models

Gianluca Pignalberi

Roberto Vacca is a Doctor of Computer Science and an electrical engineer. He is very well known in Italy because of his forecasts, mathematical and provisional models, his books (which he sells through his site www.printandread.com (www.printandread.com)) and articles. Since his forecasts, as well as his points of view, are always very sharp and are so clearly expressed, I decided to talk with him about his activity and free software world.

*What do you foresee in the future for free software?*
I think free software is the future

**Prof. Vacca, you're well known for your predictions, including foreseeing events like: the fall of the Soviet Union, and the Italian blackout in 2003. Since you don't have a crystal ball, can you explain to our readers what knowledge helps you in foreseeing such important events?**

We learn from history that empires, cultures, and organisations are created, sometimes they prosper and grow—they flourish and then fall. Other processes too go through phases and downgrading is always lurching behind us. Experience, knowledge of socio-economic mechanisms and imagination are great helps for producing plausible forecasts.

**Your predictions seem to be very much non-numerical (unlike, for example, the weather forecasts, where mathematical models are used to do "a probable prediction"). Some might even think you're just guessing! How do you determine the probability of errors in such predictions?**

I produce both qualitative and numerical forecasts. I could imagine in 1979 the downfall of the USSR on the basis of common sense. I use mathematical models to forecast (often very precisely) deterministic processes—like epidemics (deaths due to cancer or AIDS) or increase in car numbers in a given country. I can't define the probability of errors: sometimes I succeed in forecasting these numbers 15 years in advance with an error of just a few percent. Sometimes unexpected factors crop up and I am off by 50% or more.

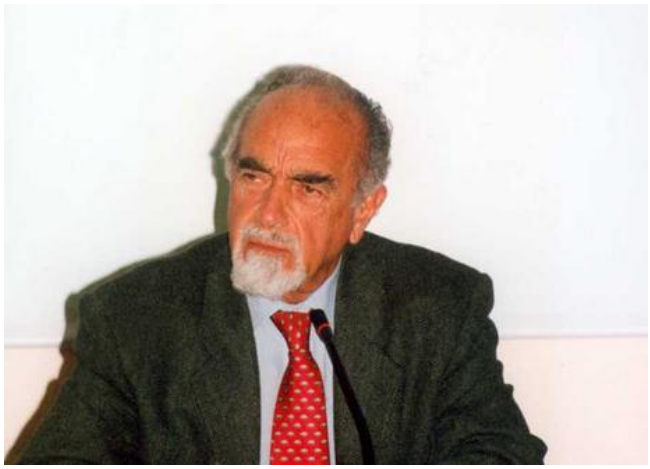
**You build mathematical models and apply them to your forecasts. Do you use computers to aid in model construction, or do you use computers to check the quality of the models or to perform simulations?**

I have developed a number of proprietary mathematical models and software packages. I use them to work out system analysis studies for my customers.

**Do you prefer using proprietary or free software and why?**

I use Word and Excel for simple accounting chores. For math analysis, modelling etc. I produce my own software—which incorporates quite a number of sophisticated tools.

Roberto Vacca, the Italian expert who forecasts events by modelling mathematical formulas and software tools



**The free software model is, roughly speaking, the following: a person, or a group of people need specific software, they don't want to use proprietary software and start designing their own tool. The resulting program and source code are freely available, so that other users can use, debug, add features, and improve on the code's quality. How do you believe this working model is good for designing software? And how, in your opinion, could it be useful in other fields, not necessarily related to computer science?**

Free software is OK, but I don't have time to document the software I produce so it can be used by others. I have only very few and occasional collaborators. So I have to take care of a lot of menial chores—and I just manage to survive in the turmoil of a very personal, extremely messy filing system. So I don't have time for niceties.

**What's your opinion on software patents? How do you believe adopting them could be an advantage, or disadvantage, for developer communities?**

I don't feel strongly about software patents one way or the other. I don't feel strongly even about copyright. For years I have published books printed by publishers (about 35 of them). I wonder whether they were very scrupulous about my royalties. For 6 years now I have been offering my books (in English and Italian) online. I give for free the first chapter of each book and the contents—if you like it, you give the data of your credit card to the secure server of my bank and you download the book in .pdf. You print, bind it (in parchment if you so prefer) and read it. Of course you can print more copies and give them away or sell them—if you are successful, it means I am VERY popular, so: good luck (mazeltov!)—but I warn you (tongue in cheek) that I'll prosecute you.

**What do you foresee for the future of free software? What advice would you give to the free software community?**

I think free software is the future. The advice is: innovate—innovate and devise crafty ways for achieving standardisation (which, very forcibly, Microsoft has done) without, at the same time, cornering the market and building an obstreperous fortune founded on software which is too heavy, cumbersome, unsafe to the extent of voiding the positive impacts and uses of innovative, faster hardware. Above all: do away with the barbaric use of icons and go back to the precise, effective, retrievable reliance on alphanumeric representations and coding.

> Innovate and devise crafty ways for achieving standardisation

I believe Roberto Vacca's interesting point of view could help the free software community build a future even better than its members are currently building. Conversely, his answers will hopefully be the starting point of a new thread of discussion.

## Copyright information



## About the author

Gianluca Pignalberi is the TeX programmer for Free Software Magazine

# 3rd EDITION
## ANNUAL CONFERENCE

**17th-19th August 2005**
Cracow, Poland

www.konferencje.software.com.pl/sqam2005

# Software Quality™
## Assurance Management

**If you test** too little, failure costs can break you. If you test too much, you risk late deliveries or higher costs than your competitors'. How can you find the "magical", optimum level? To know this, you need a lot of experience, and there is a short-cut to own experience: the experience of others. You will many experienced "others" at a good, professional conference, such as SQAM 2005.


Details:
Katarzyna Wróblewska
Phone +48 22 887 10 10
Mobile: +48 0 692 422 855
katarzyna.wroblewska@software.com.pl


**17th August 2005 – Preconference Day**

**Focus on Quality - three 2 hours tutorials concerning hottest topics**

- Agile QA - opposites attracts?
- Test Automation
- Application Security Tests

**18th August 2005**

- Keynote session (Mieke Gevers, Richard Taylor, Bogdan Bereza-Jarociński)
- A competition for the title of The Best Tester
- SQAM High Quality Product - Winners

SQAM HIGH QUALITY PRODUCT

**Workshops:**

- Test Support Tools
- Test Proces Advancement
- At the end of the day will be held evening meeting, along with the offical inauguration of the SQAM Centre

**19th August 2005**

**Lectures will be held in three parallel sessions:**

- Software Testing and Quality Management
- Software Requirements Engineering
- Software Testing Technics

sponsors:

COMPUWARE

POL SOFT

honorary patronage:

SJSI

endorsed by:

bbj Test

organizer:

software KONFERENCJE

# Games in captivity

## Liberation, emulation, and abandonware

Matt Barton

For those of us who grew up in the 80s, playing games in arcades or on our computers and game consoles was a major part of our childhoods, and we often have the nostalgic desire to replay those beloved titles. Others not only want to play, but have dedicated their scholarly attention to the study and preservation of videogame history. Sometimes companies who own the copyright to these games are able to repackage them and make them available on the shelf; there are countless "Games in a Stick" mini-consoles and plenty of "Arcade Classic" compilations for the PC and modern consoles. Unfortunately, only the most popular and well-known classic games from the biggest companies are available. Sure you can play *Ms. Pac-Man*, but what if you're looking for Paul Norman's *Forbidden Forest* or Bill Hogue's *Miner 2049er*?

There are historical, cultural, and scientific reasons to care about classic games and study them with the same devotion that old books and movies receive by venerated college professors

While many such games are impossible to find at stores, emulation enthusiasts have made them available for download from the web. Unfortunately, downloading games from an "Abandonware" site might mean breaking the law. Thankfully, people like Matt Matthews of *Liberated Games* are leading the effort to legitimize "ROM collecting" by contacting abandonware authors and copyright holders, asking them to release their games and source code under public licenses. This is an important effort with significant cultural and historical connections and major implications for future game research. This article will offer reasons why preserving older games is important, why having access to the source code is just as important as having the games themselves, and finally, why we need to do all of this legally instead of relying on abandonware sites.

### Who cares about old games?

Many people may find it odd or even laughable that some people care about preserving old, "obsolete" videogames. Why should we care if future generations of gamers are able to play out-of-production games and experience working with antiquated computer systems? More specifically, aren't games simply a useless diversion anyway? While many people may agree that preserving *Casablanca* or Shakespeare's *Julius Caesar* is a culturally significant task, taking pains to ensure that future generations will have access to *Robotron* and *Zookeeper* may seem silly. Videogames have long been a "subclass" of entertainment; something that kids do rather than what they're "supposed to", namely, their homework or playing outside. Another problem is that too many gamers view game development as a strictly linear process, with the "best" games available on the shelf today and past games as inferior or primitive in comparison. All of these factors add up to the preju-

dice that someone dedicating herself to the serious study of videogames is wasting time and resources. However, there are historical, cultural, and scientific reasons to care about classic games and study them with the same devotion that old books and movies receive by venerated college professors.

Those of us involved with the burgeoning field of game studies don't hold the view that games are simply too frivolous to be worth taking seriously. For one thing, games are an important part of our cultural history. Students in 2050 will need to know about *Space Invaders* and *Pacman* if they hope to understand the America of the early 80s, just as anyone studying the 60s will need to know about rock and roll music. One should never refuse to take something seriously just because people find it enjoyable. Videogames have become a fundamental activity for a great number of people, and ignoring them is also ignoring an important chunk of our culture.

There are other good reasons to study videogames. Many games are "deep" and have a similar emotional impact on us that great movies and books do. While many, if not most games are "me-too" rehashes of familiar formulas and gimmicks, other games explore more exciting territory. Nick Montfort identifies some games of literary weight in a brief essay named Literary Games (http://nickm.com/writing/essays/literary_games.html), but one need not look hard to find games of cultural merit. Certainly anyone who has experienced Floyd's sacrifice in Infocom's *Planetfall* or April's confrontation with her stepfather in Fun Com's *The Longest Journey* is aware that games can affect us as strongly as other mediums of expression. The *Fall Out* series gave us a vivid portrayal of civilization after a nuclear disaster, and Janet Murray has even argued in her book *Hamlet on the Holodeck* that *Tetris* is a "perfect enactment of the overtasked lives of Americans in the 1990s". Matt Matthews, host of *Liberated Games,* argues that *Missile Command* is culturally significant. "When Missile Command came along," says Matthews, "the idea of the end of the world coming in a hail of nuclear missiles was on a lot of people's minds. To see that acted out on a screen, to be in a position to try to fend this off for apparently helpless cities, to know that no matter what you did, there was no end to it—the fact that you could not avoid this fate once the missiles were launched is a political message." The point is that games can teach us valuable lessons, albeit in a vastly different way than older, more respected media have done.

The Colecovision. Pic via Wikipedia.

We could go on like this for quite a while if it were necessary. However, if we can agree that games are culturally significant, then we are likely to agree that they are worth serious study and effort to preserve them. Though some people would argue that studying any history or literature is a waste of time, I wouldn't expect a reader of *Free Software Magazine* to share such a dismal view. So, I'll move on to the more immediate issue at hand: namely, how we can access these older games, either for fun or study.

As anyone who has ever played a videogame knows, it is a very different experience reading about a game or seeing someone playing one than experiencing it for oneself. Videogaming, just like any gaming, is clearly a participatory medium that derives a great deal of its popularity from the special demands it makes on players. Unfortunately, playing older games can represent a significant technical challenge. While it's easy enough to buy a PlayStation 2 and plenty of games, finding a working Atari 2600 or an Apple II, much less games for such systems, is not. Some of us have dedicated ourselves to scouring flea markets and yard sales and preserving old games, systems, and accessories in well-kept private collections. While these collections may have cultural as well as monetary value, they are not likely to benefit many people besides the collectors who cherish them. Again, merely seeing a Colecovision in a museum is one thing and playing one is something else entirely.

Other people are not only motivated to collect classic games and hardware, but to ensure that other people are given the opportunity to experience them. One common means of achieving this end is separating the software from the hardware for which it is designed. The next step is to create an "emulator" program capable of synthesizing the original hardware. Such programs are plentiful and available for a variety of platforms. It is even quite common to find older system emulators for modern game consoles. A recent Slashdot posting (http://games.slashdot.org/

article.pl?sid=05/06/01/1215214\&tid=203\&tid=207\&tid=233) identified the plethora of emulation software already available for Sony's new handheld game console, the PSP, and Sega's Dreamcast is the platform of choice for a thriving community of emulation and homebrew enthusiasts.

## Abandonware, emulation, and ROMs

A great many, if not most, of the games we enjoyed as children and young adults are freely available for download online. Sites like *Back to the Roots, Home of the Underdogs,* and *Abandonia* provide downloads to countless games for PCs and game consoles that are no longer being sold. Thousands (if not tens of thousands) of arcade games are available in a variety of ways (including DVD mail order) for use in the well-known MAME emulator, and a cottage industry has grown up to provide hardware. Devices like the Xgaming's X-Arcade allow modern gamers to experience the feel of classic hardware. Never before in history has it been so easy to accurately emulate so many classic games on a modern PC.

> Seeing a Colecovision in a museum is one thing, playing one something else entirely

Still, even though emulation has seen significant technological progress, it remains quite illegal. The problem is that in all but a few cases, the copyright, patent, and trademark holders of these classic games and systems have not granted their permission for these downloads and are not being compensated for them. Even though most sites that host abandonware are non-profit and committed to selflessly serving communities of classic game and system enthusiasts, their activities clearly constitute copyright infringement.

The attitude of most of these sites is that they will keep a game available for download as long as they do not receive a specific request from a copyright holder that they take them down. Not many copyright holders bother to do this, and others give their blessing. The majority are probably unaware of the whole enterprise. Some abandonware authors have retained their copyright and are more than willing to release their games and code under a public license (or even dedicate them to the public domain) when they learn of any interest whatsoever in their creations. Others may not realize what is going on until their games have been downloaded thousands or millions of times; then they may feel that they have lost a great potential for revenue.

That some old games still hold significant value ought to be obvious to anyone. This is certainly the case with games like *Frogger*, *Galaga*, *Super Mario Bros.*, and *Ms. Pacman*; these games have often been repackaged and made available in a variety of compilations for modern PC and consoles. However, why should anyone buy these often expensive compilations if they can get them for free? Nevertheless, unless we're willing to break the law, we need to pay the price for these versions, since they are either being sold by the copyright holders or someone who has licensed those rights from them. Illegal downloads of classic games may seem harmless, but any attempt we make to justify doing so ends up sounding like the shoplifters who claim they steal because the store charges outlandish prices. As supporters of free software, we acknowledge the right of programmers to release "non-free" software and earn a profit doing so, even if we celebrate those who privilege freedom.

What I think we'd all like to see is more classic games being re-released under a public license so that no one has to break the law. We'd like to have the copyright holders blessing to distribute their software. However, there are a few points worth stressing here. One is very important: the original author of a game may or may not be the holder of the copyright. As anyone familiar with publishing is aware, it's a common practice that authors are required to sign over their copyrights to publishers. Once this occurs, the publisher might decide to license or sell a copyright to another entity, who may sell it yet again, and so on. In the case of old videogames, this process might have occurred many times, and unlike the system in place for patents, there is no central listing anywhere of who owns a copyright (a point of contention for many critics of modern copyright). So even authors can be totally unaware of who actually holds the copyright to their games.

Furthermore, for all practical purposes, copyrights last forever and protected works will never fall into the public domain. These are problems with the copyright system that have been addressed quite effectively by Lawrence Lessig, author of *The Future of Ideas* and *Free Culture.* Our Congress has decided that keeping Mickey Mouse from

falling into the public domain is more important than the immense treasures that might befall the public (particularly our children) were they allowed to freely distribute and build upon them. I hope that my readers will forgive me for the acid in my tone here. As a scholar and state university professor, I am perhaps more vexed than most about the inequities in modern copyright law. However, suffice it to say that discovering and contacting the copyright owners of a specific game is far more difficult and expensive than many people realize.

In some cases, giant, faceless multinational corporations have somehow become the owner of copyrights to obscure titles and may simply sit on them forever. Perhaps they bought out a smaller company long ago or somehow acquired the copyrights as part of some merger. For these corporations, freely releasing such works, even if they appear to have zero monetary value, isn't on the agenda. For one thing, the corporations aren't free to do whatever would benefit the public; they must answer to their shareholders, who might see such generosity simply as cheating them out of assets. Perhaps at some point in the future a company might pay for a license to republish some of these old games; perhaps they still retain some value as trademarks and could be regurgitated in some form for a sequel. Just because you own some property that you never use doesn't mean that you will be willing for others to build a playground on it for the neighborhood children. Even though many of us would consider this a fine use for the property and feel repulsed by a Scrooge who wouldn't allow it, we should respect his rights by law. The benefits of such laws outweigh their detriments. After all, the alternative is communism.

However, the question that so many critics of copyright like Lessig and Richard Stallman have asked is whether or not such "property" metaphors are valid. Indeed, Stallman argues that the term "intellectual property" is deliberately misleading and may prevent us from seeing these issues clearly. Such word choices cloud our reason and skew the debate in much the same way that insisting upon calling black men "boys" or women "the weaker sex" would have on a debate about universal suffrage. What we will see is that the perspective created by the use of such loaded terms often places big business interests in direct conflict with the interests of the public. To see clearly, however, we must try our best to strip ourselves of such prejudices and strive for better objectivity. When it comes to emulation, however, objectivity is a lofty goal for either side. Where passions rule, reason is exiled.

Cloanto's "Amiga Forever". Pay at the door.


One particularly inflammatory issue here involves system ROMs. A good case to illustrate this is the Commodore Amiga computer. For those not in the know, the Amiga was the platform of choice for a great many computer games of the mid-80s to early 90s. There was also a healthy Amiga public domain development community. The AMINET archives still host an amazingly comprehensive library of publicly licensed programs and data for Amiga users. However, unless one actually owns a working Amiga, which went out of production in the 90s, running these programs requires emulation.

As we mentioned earlier, an "emulator" is a type of virtual machine that emulates a particular system. Recreating another computing environment can seriously tax a system, even if the emulated system is "low-tech" or "primitive". Such software was slow and clunky in the past even for very old systems like the Commodore 64, but today's powerhouse PCs are up to the task and can faithfully emulate even the fastest Amiga computers. Once installed, emulators allow users to access software, often in binary forms called "ROMs". To put it simply, emulators allow users to run programs on their computer or console intended for other systems. However, properly emulating most systems requires access to their operating system, also available as downloadable ROMs. Without these system ROMs, emulation can't take place.

What happened with the Amiga emulation community is a chilling tale. For years Amiga enthusiasts worked to make the Amiga's public domain and commercial software library available online. The community freely shared their resources and bandwidth; there was almost an evangelical imperative to share the Amiga and its games with anyone who might want them. Plenty of people contributed their time and money to building and enriching this community.

Copyright infringement was rampant, but no one involved was making any special effort to "cash in" on the building interest in the abandoned computer and its software.

Enter Cloanto, a company that saw the public's interest in the Amiga as a lucrative opportunity for private gain. In what some might deem a sinister manuever, Cloanto secured an exclusive license to the Amiga operating system ROMs and sent cease and desist orders to anyone daring to host these critical ROMs on the internet. Cloanto was offering them for sale along with its commercial, proprietary emulation program. Cloanto made no special effort to curtail the infringement of Amiga software for which it did not own the rights, even though it is likely that anyone who bought their ROMs was engaged in such activity.

This pattern of waiting for an emulation community to form around an antiquated system, then cashing in on that community's generosity and hard work is one we are likely to see again and again. Indeed, last year there was a stir in the Commodore 64 community when a company named Tulip announced that it would soon be performing the same routine. Tulip created an "official" C-64 portal and tried to use its legal leverage as owners of the trademarks to shutdown its "competition", namely, the hundreds of enthusiasts who had been offering their time, energy, and bandwidth to the community for free. The operation did not prove profitable, and Tulip ended up selling the rights to the Commodore brand name to yet another company. The threat still lingers.

> If a law is flawed and works to society's deprivation, should we feel fine about breaking it?

What is at stake here? What's ultimately at stake is access to these culturally important computer and game systems and their libraries of programs. Legally, the copyright, trademark, and patent owners have the right to exclude others from taking advantage of their "property". The more important question from the humanist's perspective is whether such pragmatic business considerations trump cultural or social concerns. Legal or not, should we deprive future generations access to culturally significant videogames for the sake of a *potential* profit for some unknown corporation?

This question is particularly tricky given the internet and the nature of software. If I decided to setup a printing press and start publishing novels that were out of print yet still protected by copyright, my activities would be highly visible. Word would get back to the copyright owners, and I'd likely be sued. Just making a living selling these novels would adequately demonstrate that the works were still commercially viable and that I was "stealing" from the copyright owner by taking advantage of them. Even giving them away for free would not make me immune to prosecution. After all, if there is enough demand to stimulate free publication, mustn't there necessarily be commercial demand as well? If a hundred people accept a book for free, perhaps one or two might also pay for it. Some would try to apply this same example to sharing ROMs online; if a thousand people would download a ROM for free, might not some of them also pay for it?

This argument seems silly upon serious reflection, yet it is this same kind of thinking that underlies our copyright laws and makes generosity and sharing vices rather than the highest virtues. Anyone with sufficient intelligence and concern for his fellow citizens is aware of how badly we need to reform our copyright laws. However, let us not dwell too long on these issues here, since they are explored so much more fully in so many other works. The works of Siva Vaidhyanathan, Richard Stallman and Lawrence Lessig make an exceedingly compelling case for such reform and deserve a place on every good citizen's reading queue.

## Liberated games

Abandonware enthusiasts have frequently argued that in the case of old games, we ought to disobey the law. After all, copyright law is no longer serving its purpose and has been warped to serve only the interests of multinational corporations instead of authors or the public. If a law is flawed and works to society's deprivation, shouldn't we feel fine about breaking it? Others contend that while "pirating" a new game is wrong, copying and distributing "abandoned" games is ethically justified. In fact, the term "abandonware" is itself a loaded term; the idea underlying it is that a commercial entity has abandoned the product and thus we ought to feel justified in taking and using it ourselves. After all, if someone leaves a kitten to starve to death on the side of the highway, we would feel quite justified in taking the kitten home and caring for it properly. Shouldn't the same be true of software? Furthermore, even if the original owner

showed up later at our door demanding the cat back, would anyone condemn us for telling that person where to go? The strategy behind the term "abandonware" is this idea: if the software has been "abandoned", then the owner no longer has any rights over it. It's free for the taking.

Matt Matthews, creator of *Liberated Games,* takes issue with the practices of abandonware sites that offer copyrighted works without the explicit permission of their owners regardless of whether the works are commercially viable or not. Matthews' argument is that since these sites are breaking the law by offering copyrighted works for public download without first acquiring the permission of the copyright owners, we ought to avoid using them. This position might seem untenable for someone running a website called *Liberated Games,* another loaded term with obvious rhetorical implications, yet Matthews argument makes sense when given more context.

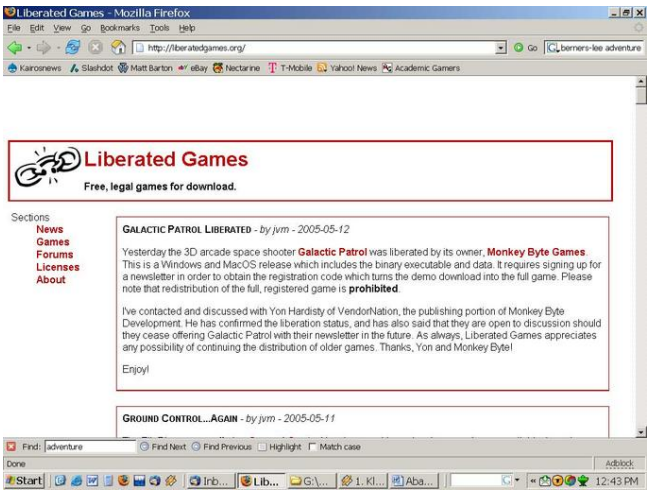
> No other medium of human expression offers the kind of "peeking under the hood" that source code provides

Matthews wants to encourage the owners of copyrighted "abandonware" to explicitly release their works into the public domain or under a public license. Furthermore, he wants to see them contribute the source code along with their binaries. Matthews will not host a ROM or other download if he cannot verify that the copyright owners have explicitly made it legal to do so.

Abandonware enthusiasts might simply dismiss Matthews as a sort of "Dudley Do-Right" character so concerned about following the letter of the law that he is blind to its inequities. Matthews' argument, however, is that offering works without acquiring permission is unfair to the owners of the copyrights and brings the whole abandonware scene into disrepute. Not all of us who want to emulate old games on our computers want to be associated with "outlaws". Matthews goal is to make games available for download that are "in the clear" and unambiguously free and legal for download.

Abandonware sites like to defend their activities by claiming that the games they offer have no commercial value because they are not in production and unavailable for purchase. Matthews compares the situation to Disney's feature

Screenshot of Liberated Games




films. Disney's practice is to withdraw a movie for a few years, let the market pressure build up, then re-release it for a limited time. Matthews argues that it would be wrong to assume that just because these movies are temporarily unavailable we should feel justified in copying and distributing them without authorization. "It is not my position or anyone else's to decide whether there is commercial value there or not," says Matthews. "It's the copyright owner's position to decide that." Disney intentionally limits the availability of its movies to maintain the value of its copyrights. According to Matthews, we have to respect Disney's decision and not take it upon ourselves to break the law no matter how badly we want our children to see *Bambi*.

Matthews' goal is not to police abandonware sites or chastise people who use them. Instead, Matthews wants to draw our attention to the copyright holders who have chosen to release their works to the public. Clearly, these people are doing us a great service, and we ought to encourage this behaviour. If we limit our abandonware collections to strictly those works we can acquire legally, we may help persuade other copyright holders to release their works. How could a copyright holder of an old videogame benefit from releasing that work to the public? There are several very real potential benefits. Firstly, there may be a buzz about the released game on message boards and retrogaming sites. This might renew interest in the game and provide a sort of free advertisement for a developer or brand. This seems to be the motivation behind Rockstar's decision to release *Grand Theft*

*Auto,* or Id's many contributions. Sierra released (temporarily and with restrictions) its game *Betrayal in Krondor* to drum up enthusiasm for its sequel, *Betrayal in Antara.* Another motivation might be simply to ensure one's spot in gaming history by making one's work as widely available as possible. A good game that is free to distribute is likely to survive longer than one that can only be traded illegally.

Another reason for contacting the copyright holders is that we might also acquire source code. Matthews argues that it's not enough for copyright holders to release the binaries of their games. They should also release the source code. "It's my opinion that if you're going to give away the game for free, I don't see how it is any more of a loss to give away the source code—and I see all kinds of benefits to that." One of these advantages if that third parties might use the source code to port games to other platforms, such as the *Doom* and *Quake* ports for the Dreamcast. "I don't think there's any question that a programmer developing for embedded hardware, where space and RAM are small, could learn from code for the Atari 2600." Especially in cases where the author is also the copyright holder, we have opportunities to find and release valuable source code.

A question I recently posed on academic-gamers.org (http://www.academic-gamers.org/discuss.shtml?storypath=/main/blacklily814.html\&flav=daisy) asked game scholars whether they would benefit from having access to source code. Admittedly, not all game scholars are programmers or have more than a superficial understanding of what goes on under the hood of the games they study. While I would draw the line at saying that studying the code will *necessarily* lead to a better understanding of a game, I can clearly see how it could lead to insights otherwise unattainable. As Laurie Taylor points out, "Coding constraints—which in the past often necessitated odd coding structures and the repetition of certain portions of code—can aid in the study of game design itself; for instance in the repetition of certain spaces, gaming tropes, in the structure of gaming music, and so on". No less a computer scientist than Donald Knuth has explored the source code for the legendary *Adventure,* using his CWEB programming language to help readers appreciate its brilliance. I can imagine a future where students not only study the games themselves, but also the code, and thereby gain an even better appreciation for the programmer's ingenuity and insight. After all, studying a game without its source code is comparable to studying music without any knowledge of theory or painting with no inkling of technique. Even these examples do not quite get the point across, because no other medium of human expression offers the kind of "peeking under the hood" that source code provides.

## Games in captivity

In this article, I've discussed several problems faced by classic gaming players, preservationists, historians, and scholars. From a technological perspective, emulating and thus allowing future generations to experience classic games is not a problem as long as we disregard the significance of playing a game on its original hardware. By far the biggest problem is the legality of emulation and abandonware. Although the restrictions imposed on society by what have become tyrannical copyright laws are experienced and often resented by scholars who study books or films, they are particularly unbearable for game scholars. Unlike the world of books and movies, there are no videogames in the public domain apart from those specifically released as such. Any book or film published before the Great Depression is free for the taking, but even the oldest known videogames fall under the protection of copyright law—unless those who hold copyright have specifically disavowed these rights.

Furthermore, although there are many commercial and non-commercial games that have been released for free public distribution, there is still a dearth of source code. This is unfortunate when we think about how often game programmers have been the pioneers of the entire software and hardware industry. Programming history is full of examples of how an innovative technique intended for a game had consequences for the industry. The history of Unix is a prime example. The innovations that led to this vastly influential operating system were a result of Ken Thompson and Dennis Ritchie's efforts to play his game *Space Travel* on a PDP-7. Bell Labs' own History of Unix website describes the significance of this early effort to play a game intended for one platform on another: "Their effort included a floating-point arithmetic package, the pointwise specification of the graphics characters for the display, and a de-bugging subsystem that continuously displayed the contents of typed-in locations in the corner of the screen". Would we have

Unix or Linux today if it weren't for *Space Travel*? We certainly don't have to spend time arguing about the importance that games had for multimedia. It wasn't so long ago in IBM-PC history when monitors were monochrome and sound was limited to a tiny speaker (or "buzzer") inside the PC. Finally, Tim Berners-Lee has acknowledged the influence of the old game *Adventure* for his development of the world wide web (see Berners-Lee's Information Management: A Proposal (http://www.w3.org/History/1989/proposal.html) for more information). A history of programming that failed to consider the significance of gaming is simply unthinkable.

I would like to see more classic games and their source code legally available for public access and distribution. This will only happen if more of us take Matthews' lead in eschewing illegal methods of accessing them. Instead downloading ROMs illegally, we ought to be joining the effort to contact the copyright holders and liberating these games. Perhaps we can form a consortium dedicated to securing or buying up these "worthless" copyrights and promptly releasing them along with their source code under a general public license. What better way can we honor the developers of our favorite games than by asking them for permission to share and celebrate their works far and wide, at our own expense?

I have merely touched here on issues that could easily be discussed in book length. My hope is that the article may prove of some use to you in forming or considering your own views on this subject and that you will share your thoughts with me in the comments section of Free Software Magazine's website (http://www.freesoftwaremagazine.com).

## Copyright information



## About the author

Matt Barton is an English professor at St. Cloud State University in Minnesota. He is an advocate of free software, wikis, and the Creative Commons. He also studies and writes about videogames.



# Emulation

## Bridges over troubled waters

Matt Barton

The term *emulation* means to either equal or exceed something or someone else. As computer jargon, however, *emulation* means recreating another computer or console's operating system on another system; e.g., recreating a Nintendo Entertainment System on your Sega Dreamcast so you can boot up a *Super Metroid* ROM, or playing classic arcade games like *Ms. Pac-Man* or *Omega Race* on your Gameboy Advance SP. Certainly, neither Nintendo nor Sega ever meant for their systems to be used for such purposes.

The obvious utility of emulation software is that it allows users better compatibility and more freedom. The obvious threat of the software is that it allows users better compatibility and more freedom.

This article is not a "how-to" or a guide for emulation. If you want to learn how to emulate a NES on your Dreamcast, there are better sources of information. What I'm concerned with here is why we, as promoters of free software and concerned citizens in general, should care about these issues. As I will endeavour to show, emulation is not just about playing the latest PS2 title on your PC or even allowing new gamers access to older titles. It's about the freedom to run the software you want on the hardware you want.

### The threat of emulation

As we discover so often in the computer industry, emulation involves a clash between private and public interests. It is in the best interest of most software and hardware corporations to "lock-in" users. If you want to play a Nintendo game, Nintendo wants you to buy a Nintendo Entertainment System. If you want to play *Metroid* on your GBA, Nintendo expects you to pony up $20 for the "authorized" version they're selling for the GBA even if you bought and still own the original cartridge for your NES. (It is quite possible to emulate the original on your GBA.) If you want to run Amiga Software, you'll need an Amiga, and a company named Cloanto is ready to sue you if you don't pay them for the privilege of emulating the system on your PC. Microsoft's strategies to shut out GNU/Linux users or impose Internet Explorer on web surfers are well known. Emulation represents a threat to corporations who thrive on the exclusivity of their software and hardware.

What's the big deal? Who cares if you're not free to emulate other systems on your PC or console? Probably the most obvious reason I care is convenience. As someone who enjoys researching game and software history, it's neither convenient nor economically viable for me to own all the hardware I need to run more than a few systems. To use an analogy that Larry Lessig uses for a different purpose, imagine if the highway system were built so that only a particular make of car could drive on a particular kind of road. With such a system in place, you'd need three different automobiles to get from California to Florida and two more to get to New York. If one car could "emulate" the rest and get people around without the need for separate vehicles, everyone would want one no matter the "threat" such a car would pose to the other manufacturers. Sure, perhaps

a Toyota would perform better than a Mazda on a "Toyota Highway", but as long as I can emulate my way to Bourbon Street in my Miata, I'll deal with it.

Emulation has been with computers since the dawn of computing, and it certainly hasn't always carried the stigma it does today. In 1965, IBM released its new IBM 360 line. The only problem was that IBM's customers would rather live without the bells and whistles of the 360s if they could continue using the same software they'd paid so much to develop and spent so much time getting familiar with. IBM anticipated their fears and made the switch a no-brainer. Along with their 360s came an emulation program by Larry Moss (http://www.zophar.net/articles/art_14-2.html) that allowed users to run software written for IBM's older 7070 systems. IBM realized quite rightly that users would be reluctant to upgrade their computers if it meant reprogramming or purchasing a whole new suite of customized software. Moss's emulator made switching less costly, and was thus quite popular with IBM's clientele. Moss's emulator was IBM's trump card, and they won big with it.

If every emulator was as helpful to a company's bottom line as Moss's was for IBM, there would be no "threat" of emulation. Every corporation would welcome them and probably develop emulators themselves. As long as emulation software is programmed "in house" and kept under corporate control, it's a jewel in their crown. The problem is that it is just as easy for a competitor to take advantage of emulation software, which can become a powerful weapon indeed if a company is relying on the exclusivity of its software to maintain its market share. This was certainly the case with Colecovision's "Atari 2600 adapter" released in 1982, which gave owners of Colecovisions the freedom to select from two formerly competing libraries of games. There's no good reason to buy one of Atari's game consoles if you can play all the Atari games on your Colecovision—*and* play those near pixel-perfect Colecovision arcade conversions you've been hearing so much about.

There is also a concern that emulation may be bad for current hardware; ensuring backwards-compatibility may limit a system or drive up manufacturing costs. Commodore made no serious effort to promote C-64 emulation when it released its newer Amiga line and didn't even include a $5\frac{1}{4}$" floppy on any of its new machines—a decidedly different strategy than it had taken with the C-128, which could switch between three different modes: C-128, C-64, and C/PM. Sony's decision to incorporate backwards-compatibility with the PS1 in its PS2 was seen by some as a concession to consumers, but by others as a rather meek strategy to buff up the game library for their new console (a few hundred PS1 games looks a lot better than 3 or 4 rushed PS2 titles on release day).

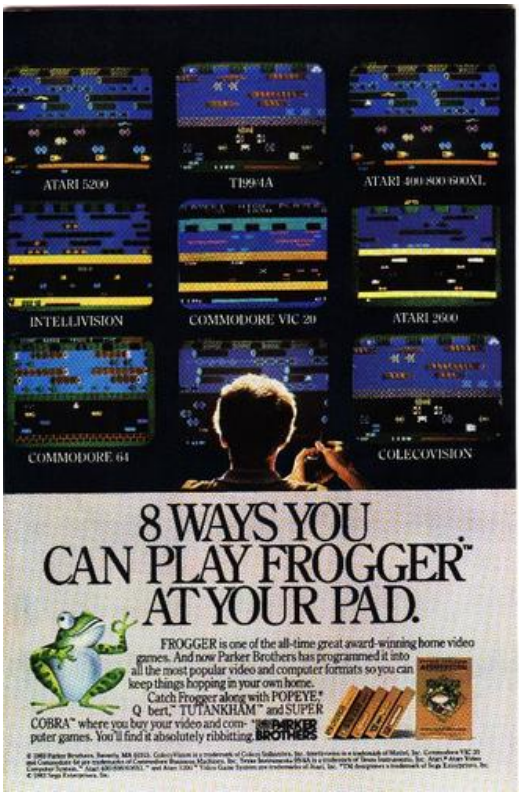
Another economic issue is that users may not want to buy new accessories like joysticks or mice or new versions of old software if they can use the ones they purchased for another system. Console manufacturers want to do everything they can to increase "customer loyalty" by forcing people to purchase proprietary accessories that will only work with their systems. Those of us who can still remember unplugging our Atari 2600 joysticks and plugging them into our Commodore 64s (and later our Amigas) are just too spoiled to tolerate the dozens of incompatible plugs and ports on later consoles.

Yet we'll still buy a console if it has "the game" we can't live without. Console manufacturers and operating system developers know that if a very popular software program is only available for one operating system—the "killer app"—then consumers will purchase it regardless of the sophistication of another machine. No one knows this better than Nintendo, who have milked their various "franchises" to the point of absurdity (what's next, *Yoshi's Love Match Timber Sports?*). Consumers will also be drawn to computing platforms with the most software. In the 80s, consumers were often bewildered by the dozens of incompatible computer systems on the market. Computers like the Commodore 64, Tandy TRS-80, and Apple II competed savagely with each other and divided consumers, who often became fiercely loyal to whatever brand they purchased. Companies who dared to manufacture "clones" of these proprietary machines, like Franklin, were taken to court and driven out of town. The money was assumed to be in proprietary hardware—an assumption which Microsoft wisely did not share. The war continued into the "16-bit" era, and it is still easy to find diehards still clinging to their proprietary Amiga or Atari computers despite their alleged obsolescence. When you're asked to learn another operating system from scratch, it's amazing how superior your old one feels.

Game consoles underwent a similar turf war in the 80s. Seemingly every major department store was offering its own "television game". Few families could afford more

than one system, so the games that children played were limited by whether their living room was home to a Colecovision, Intellivision, Atari 2600, Adventurevision, Astrocade, or Odyssey II—to name a few. Each system had its special features, quirks, and a game library that distinguished it from the competition. Unlike the personal computer market, the hardware manufacturers produced nearly all of the software "in house" or under exclusive agreements and even tried to hide the identity of their programmers to protect against competition from other console manufacturers. When four programmers (including David Crane, later developer of *Pitfall*) left Atari to begin producing unauthorized games for the 2600 in 1979, Atari promptly sued to protect its monopoly (a strategy that would be later repeated by Nintendo against Atari, who'd try the same thing). The stakes were high and Atari was too short-sighted to see the advantage of third-party developers in helping it to sell consoles. Atari eventually lost in 1982, the same year Colecovision released its "Atari 2600 Adapter", which allowed Colecovision owners to play any of the Atari 2600's titles. Atari sued and lost yet again. 1982 was a year when consumer interests mattered to our elected officials, but in 1983 came the great videogame crash. So it goes.

80s *Frogger* ad showing screenshots from 8 systems. From Tomorrow's Heroes website




Whenever there is an abundance of platforms, developers have to port their programs to a wide variety of machines if they want to make the most profit. A glance at the ads in many comic books of the 80s reveals a montage of screenshots of the same game across systems. Though these ads are intended to sell games rather than game systems, it is clear enough which systems have the best graphics. Thus, hardware manufacturers could no longer offer inferior products in the hopes that their exclusive software line up would make the difference. Now they had to court not only consumers but developers—if they could not force developers to release a title only for their system, then perhaps they could coax them with more sophisticated hardware. In a trend that continues to this day, the popularity of any given game console relies at least as much on its third-party developers than the sophistication of its hardware. Software developers want to reach the widest number of consumers; hardware developers want to limit the availability of software to their own systems.

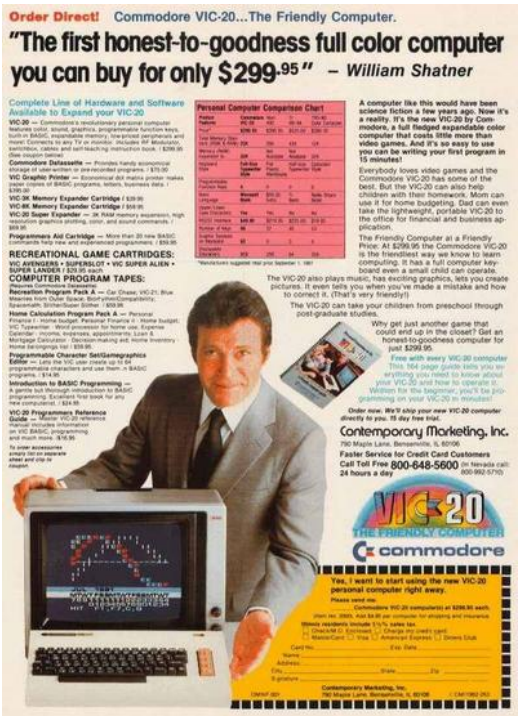
The three parties involved: hardware developers, software developers, and users; each with their own interests, most of which conflict with one or the other. It's fun to think about the changes that would come about if any one of these groups dictated the terms. Hardware developers want to make software exclusive to their products so that users who want the software must invest in their hardware. They make their products as incompatible with the competition as they can. Software developers want to reach the maximum number of consumers with the least amount of compatibility issues, so they push towards generic *hardware* standards while closing up *software* standards. Meanwhile, users want open hardware and software, since the market rewards consumers when the maximum number of competitors is allowed to participate—a situation that arises in periods when monopolies do not exist. Ultimately, a user doesn't so much want an "Apple"; a user wants a certain set of hardware and software features that are "Apple's" only by virtue of patents, copyrights, and trademarks. A consumer would like to buy a trackball or joystick without having to worry about whether it'll work on a particular machine.

If the world of software development was truly autonomous from the world of hardware development, emulation would

not be an issue. The reason it remains a problem is that hardware manufacturers (and the developers of non-free operating systems like *Windows* or *Tiger*) have proven themselves willing to enter into lucrative licensing agreements with the makers of popular games and applications time and time again. Sony and Nintendo fought hard for the *Final Fantasy* series, and it's not unthinkable to suggest that Square's decision to go with Sony ended Nintendo's stranglehold on the console market of the 90s. Similar examples from personal computing are also plentiful. Clearly the Commodore Amiga would have died off years before if NewTek's *Video Toaster* was also available for IBM-compatibles or Apple Macintoshes. Apple defended its graphical operating system fiercely, even suing Microsoft for daring to call its "Trashcan" feature by the same name. Microsoft, the unabashed king of operating system lock-in, helped to destroy hardware vendor lock-in by convincing IBM to use Microsoft's operating system on their computers instead of developing their own. Once Microsoft had its foot in the door, it was all too easy to encourage clone-makers and third-party developers to swear their fealty to the new standard. The software industry was now wagging the dog, and IBM's OS/2 failed to reverse the situation. Clones running DOS and later Windows won out because a platform that can do all things adequately trumps any other that can do fewer things better.

It's this last fact that brings us to the threat emulation software poses to the software industry. Compared to the threat emulation poses to their livelihoods, unauthorized copying and distribution are the flies distracting them from the lion. If I can successfully emulate a propriety operating environment on a free machine, then there is no reason to own the proprietary system, which will always be more limited than the superior one capable of emulating it. If I can play Nintendo DS games on my Sony PSP, but not vice versa, then the choice is obvious which system I should purchase. If a generic portable like the GP32 (http://en.wikipedia.org/wiki/GP32) can emulate both, I won't even bother with the proprietary models. Likewise, if my powerful PC can emulate all old and modern consoles flawlessly (or even improve on their performance), then I would be foolish to invest in a game console—to paraphrase Will Shatner's old slogan for Commodore, "Why buy a system when you can have an emulator?". We shouldn't wonder why Nintendo has taken to experimenting with control

Commodore Vic-20 ad featuring Will Shatner.




and interface and introducing bizarre "innovations" in these areas: they are worried about emulation.

Is Nintendo right to worry? Perhaps. In 1999, an emulator for the 3-year old Nintendo 64 was released—UltraHLE. UltraHLE demonstrated that it was possible to successfully emulate *modern* consoles at a playable frame rate on reasonably equipped PCs. Emulators would soon follow for other modern consoles. To put it crassly, the emulation scene "hit the fan" when UltraHLE debuted and lost whatever innocence it had hitherto purported to possess. Emulation software could now be used to copy and distribute games currently on the market—it wasn't just about obsolete systems anymore.

Many emulation enthusiasts have long claimed that they are not interested in "piracy" or promoting the unauthorized distribution of commercially viable programs. A typical enthusiast might claim that, though technically illegal, it is ethically justifiable to swap old titles that are no longer being sold commercially, but doing the same with modern games is wrong. If I can't buy *Gorf,* then what's wrong with downloading the ROM? Such an attitude is untenable. We cannot at once recognize software as both private (legitimizing pro-

prietary software) and public property (legitimizing "abandonware"). The law makes no distinction between copying and distributing old or new software, since no software is in the public domain unless its copyright owners officially have officially declared it so.

Nevertheless, the economic threat emulation poses to modern consoles seems negligible. A PC powerful enough to emulate any of the "Big 3" would be exponentially more expensive than simply buying the console. As long as there is a significant price gap between the cost of a given console and a machine with enough power to emulate it, the threat is marginal. Ostensibly, someone with thousands to spend on computer hardware would view modern console emulation more as a curiosity than a practical replacement for his game console. We talk of "hundreds" spent on console gaming, but "thousands" spent on computer gaming. There is no threat here.

## The challenge of emulation

Let us hubristically assume for the moment that a free operating system had been built that could, for a reasonable investment in readily available hardware, not only emulate Microsoft *Windows* but actually run it faster and more efficiently than is currently possible. If it is possible today to emulate a Commodore 64 or Amiga at exponentially faster speeds than those computers were capable of reaching, it seems possible to do the same for the modern PC on a significantly more advanced machine. I say "hubristically", because of course such a feat would entail some fundamental discovery of computer engineering far more exciting than the mere execution of a Microsoft application. Emulating another system is difficult because it requires a system to not only support its own environment but also recreate another within it. There is an order of magnitude difference here in terms of complexity. In short, there is an exponential relation between the complexity of system X and system Y's ability to emulate it.

Consider a simple example. Say that you've been chosen to play the role of a famous surgeon in an upcoming documentary. To prepare adequately for the role, you'll need to not only learn your lines, but study the surgeon's actual techniques and thus offer a convincing performance. You can imagine being asked to spend several days observing surgeons at the local hospital, and then spending time in front of a mirror attempting to "get it right". Anyone who has spent any time with professional actors knows how much hard work goes into preparing for a serious role. We might compare this type of acting to "simulation", or a program known as a "port". It's not *really* the software you wanted, but it fools you into thinking it is as long as you don't peer under the hood.

However, you haven't been asked to merely *act* like a surgeon, but to actually *be* one. In the film you will actually perform a real surgery—but after it's all over, you can go back to being an actor and prepare for your next film, perhaps one in which you will not only play a nuclear engineer but *be* one. No human actor could possibly undertake such tasks—by the time they'd been through medical school and all the intermediate steps required to be a surgeon, they'd be too old and otherwise preoccupied to worry about a silly film!

While not perfect, this example does give some idea of what a system is asked to do when attempting to emulate other systems. We're asking our operating system to not only be itself, but also, at the same time, to be something else. With this in mind, it's easy to see why the "emulation scene" is mostly concerned with antiquated consoles and computers and why it takes a powerhouse of a PC to emulate even systems like the Amiga.

Emulation programs for antiquated systems are available for both computers and consoles. Perhaps more importantly, the "ROMs" of games for these systems are widely available on the internet. "ROM" is the "street name", if you will, for a "ROM Image Definition". Obviously, you can't just plug an NES cartridge into your computer (another advantage of the cartridge system for the game industry). However, hackers have long ago found ways to stream the data from a cartridge and store it as a binary file called a "ROM image". This ROM can then be loaded into an emulator. Of course, it didn't take long for these ROMs to start showing up online, where emulation enthusiasts can download them for free.

Andrew Wolan, manager of the popular emulation website *EmulationZone.org*, argues that the internet has "helped speed the [emulation] process by collaborating efforts worldwide", bringing emulation from the workshops of a few hardcore hackers to the mainstream. Sega's defunct Dreamcast system is home to a thriving emulation scene, as are most modern consoles (even the portables). GNU/Linux

GP32 portable gaming device. Pic from Wikipedia.

is home to countless emulation packages—indeed, many of the most popular Windows emulators are ports of programs created for GNU/Linux. The Multiple Arcade Machine Emulator, MAME, is available for a wide variety of computers and consoles. MAME is an especially impressive achievement in emulation because arcade machines were often even more "proprietary" than game consoles, because they only needed to run one game. The hardware could thus be built especially to support that one game. Thanks to MAME, those lucky enough to own a powerful PC have access to nearly every arcade game ever created and can emulate classic computer systems at speeds never dreamt of by their designers. Never before in history have so many programs been available for a single platform. How many of us have our hard drives loaded with thousands if not tens of thousands of games for dozens of systems; some of which we will never have the time to try once, much less play through?

## The future of emulation and free software

Perhaps the most significant implication of emulation software is that it offers users more platform independence than they have ever enjoyed before. Armies of hackers and hordes of enthusiasts have used the internet to accomplish what the industry would never have given them—the freedom to run the programs they want on the platforms they want. If I want to play *Donkey Kong Country* on my PS2, I can. If I want to run an Amiga desktop on top of XP, I can. The choice is no longer up to hardware and software developers. Eventually, it seems likely that these boundaries will blur to the point where the term "platform" is no longer applicable. Generic devices will run generic software, and the equivalent of the operating system will simply be a "smart" emulator that will automatically configure itself to accommodate a program. Of course, such ideas are nothing new; Java is intended to fill much the same niche, though it has not caught on as well as Sun had hoped.

Perhaps the problem is that we have yet to see a significant "open source" or generic movement in hardware development to equal the achievements of software programmers. The reason for such a lack is not hard to see. While some PC components are generic and easy to find cheap (mice and keyboards, for instance), designing and manufacturing sophisticated hardware like CPUs and graphic cards is a much more expensive process than software development—only major corporations like Intel have the necessary funds. Most of the components in our modern PCs are not fundamentally different from IBM's original model, and modern consoles seem to be moving towards the same architecture. Though many of us are impressed with the gains hardware has made over the decades (particularly in graphics, processor speed, and so on), we are talking about innovations to an existing model rather than the invention of new models. How many refinements will we make to our steam engines before someone discovers the potential of gasoline and catapults the industry into the next age? Indulge me for a moment. The kind of "super computer" that will be capable of the high-end emulation I have in mind is not possible without radical new (and open) hardware. Such a computer would have the power to emulate modern systems like GNU/Linux and Windows while at the same time offering exciting new applications previously impossible. If high-end PCs struggle to reach speeds of 3 gigahertz, this machine will run at terahertz. Furthermore, it will not be tied to any particular manufacturer of hardware or software, but remain open so that manufacturers will compete and the market will work to lower production costs. Eventually this machine's emulation possibilities would destroy the commercial viability of any platform-specific development model and all software would be platform independent by necessity—there would be no platforms!

Such dreams are often derided by cynics, who are always on hand to offer compelling arguments or observations about

why the status quo is here to stay and true change is impossible. George Airy, an important statesman for science during Babbage's day, convinced most of Britain that mechanical devices for computation could *never* excel the speed or accuracy of human calculators working with slide rules and tables. If there is one thing that studying the history of science has taught me, it's that "impossible" simply means we haven't figured out a way to do it yet. Often enough, what holds us back from solving a problem is that we keep asking the wrong questions or making the wrong assumptions. We assume, for instance, that tomorrow's computers will be faster because the CPUs will be faster, not that the very architecture of the PC will be rebuilt or that the thing we now call a "computer" will not be a grey box but built piecemeal into the other devices we'll carry around with us. We need to be listening to revolutionaries on the frontier, not Intel and Microsoft's corporate speak.

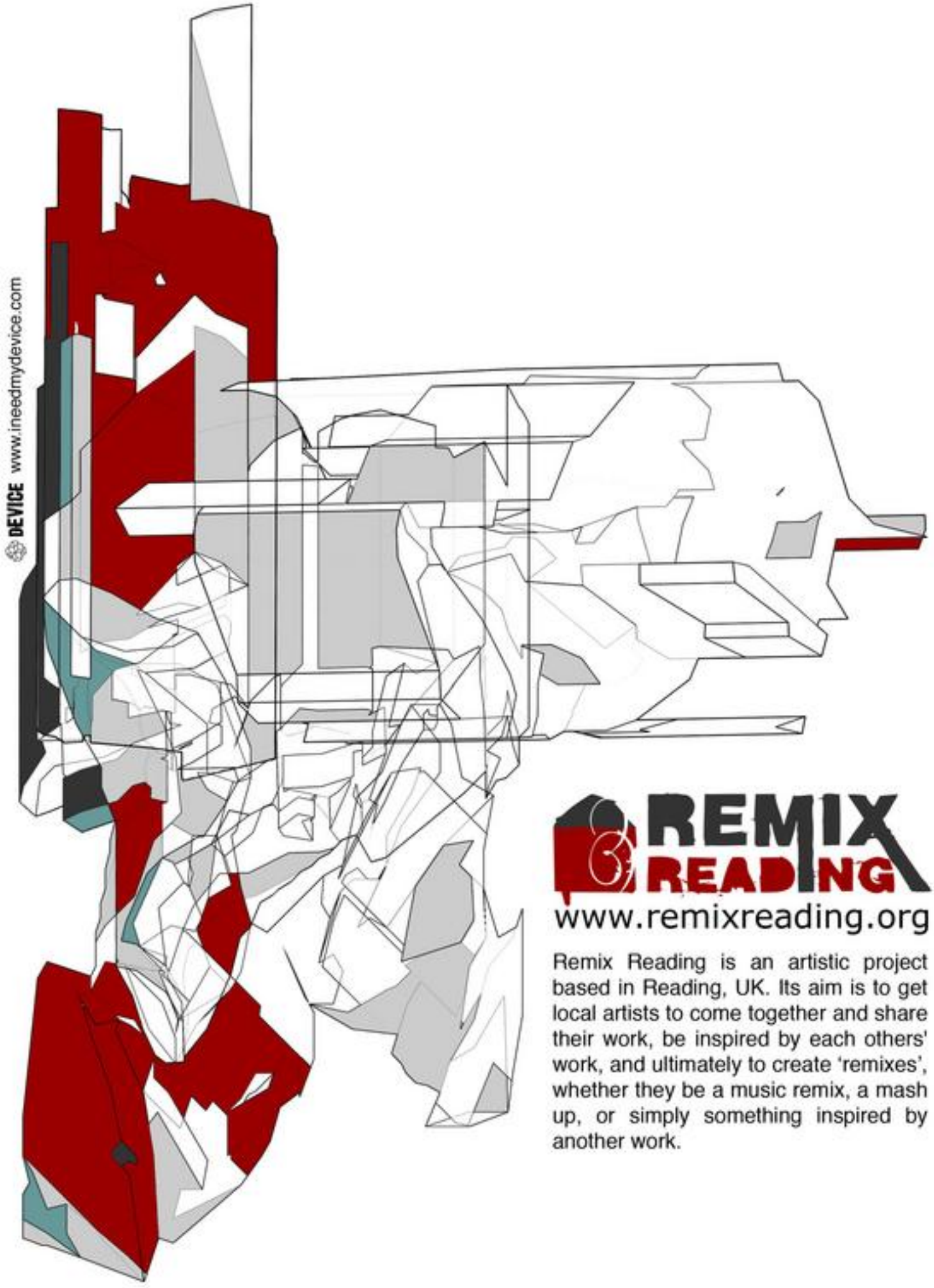
Those of us who are eager to not only push the computer industry forward, but to maximize its potential to make citizens of consumers, need to stay focused on emulation. We need to add our concerns about emulation to the attacks we make on tyrannical laws like the DMCA, which make reverse-engineering (a critical aspect of emulation programming) a federal crime. We need to build our free operating systems and open architectures so that they not only run the best of free software, but are also able to emulate whatever the proprietary world has to offer—for once a proprietary system is successfully and cheaply emulated on a free platform, revolution is inevitable.

## Copyright information



## About the author

Matt Barton is an English professor at St. Cloud State University in Minnesota. He is an advocate of free software, wikis, and the Creative Commons. He also studies and writes about videogames.




# REMIX READING

www.remixreading.org

Remix Reading is an artistic project based in Reading, UK. Its aim is to get local artists to come together and share their work, be inspired by each others' work, and ultimately to create 'remixes', whether they be a music remix, a mash up, or simply something inspired by another work.

# Replacing proprietary anti-virus software

## Using ClamWin Free Antivirus to replace proprietary anti-virus software

Robin Monks

One of the major software programs we should be using every day, is a virus scanner. This single piece of software can be found on almost every PC in the world.

It is also a major source of funding for companies like McAfee and Norton. So I was pleasantly surprised when I got a note about a new free software alternative to these costly proprietary anti-virus programs, ClamWin Free Antivirus (http://clamwin.com).

ClamWin has gone so far as to win the "Project of the Month" award (http://sourceforge.net/potm/potm-2005-02.php) from SourceForge. This month, I'll happily guide you through ClamWin's strengths and weaknesses and guide you through some key features.

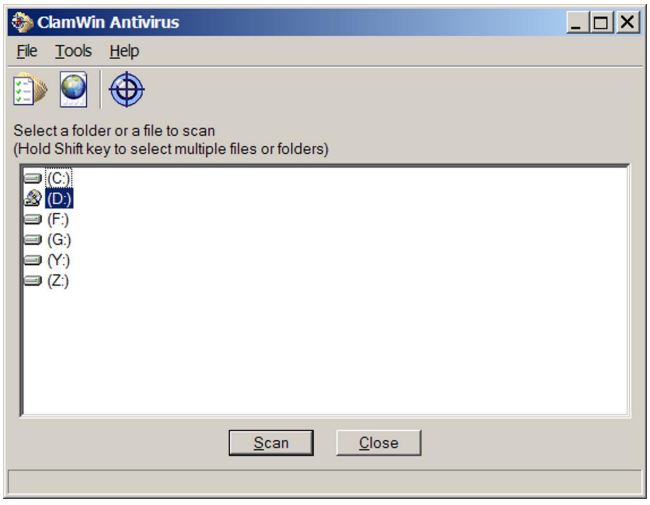
### Installation and configuration

ClamWin has a nice, easy to browse website which quickly lead me to the download. At under 5MB it was a breeze, even for my poor old 56k modem.

The installation went without a hitch, taking only a few minutes to install the software with the default options. The installer also lead me though a simple update process, which went by quickly. This process gets the latest definitions from ClamAV (http://clamav.com).

The program was installed fully configured, although many options are available for more advanced users. There's also an Outlook email scanner and context menu add-in for Windows Explorer.

ClamWin's main screen




All in all, I had the program set up in 5 minutes (minus the update process). After completing the setup process the application immediately started in my system tray.

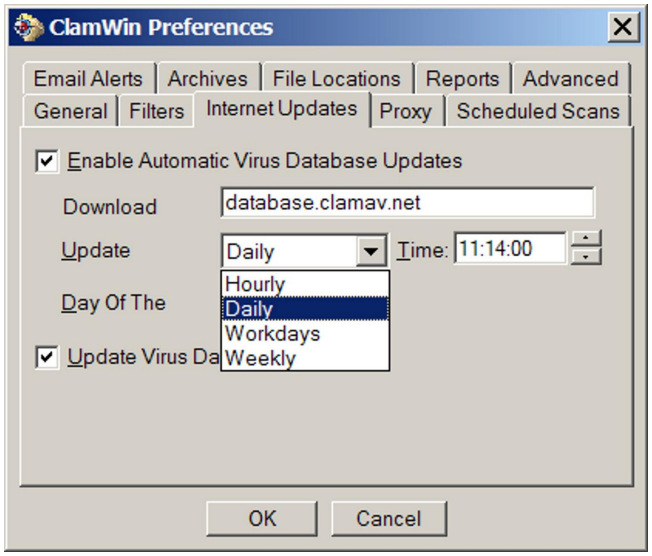
### Starting up

ClamWin provided a very user friendly interface and ample documentation.

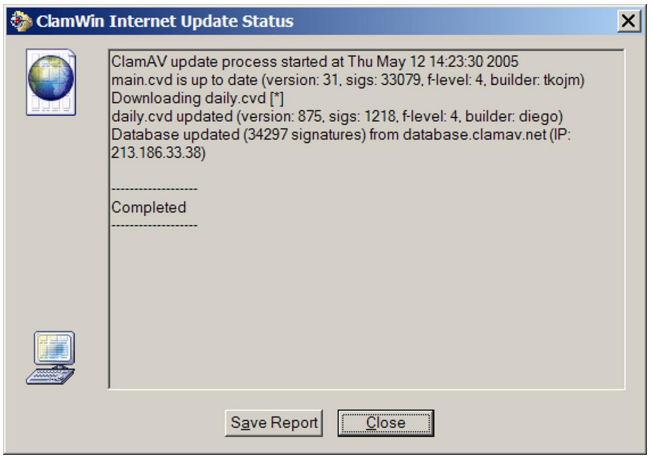
The entire program was well laid out and all of the main features could easily be accessed from the main window.

It's important to note that ClamWin doesn't perform active scanning (e.g. Scanning files as you open them). ClamWin

ClamWin's preferences




Updating ClamWin




requires you to perform scanning manually or via preset scheduled scans.

The installation went without a hitch, taking only a few minutes to install the software with the default options

ClamWin also allowed me to set up automatic updates. In my testing I didn't check the typical update time, but it should be rather short as only the definitions need to be updated.

## Scanning

I tested ClamAV by scanning a 221MB CD-ROM. In my tests it took 6 minutes (noting that the proprietary McAfee takes only 30 seconds). The scanning interface was very clean and uncluttered. I was unable to test with real viruses (for obvious reasons), however the ClamAV libraries are very trusted, and hence should pick up any nasties.

The context menu scanner also works very well, allowing you to quickly scan any files you might have any doubts about. The email scanner only works with Outlook, although files can be scanned individually outside the client.

The scheduler can be used to set scans and updates to occur when you're away from your computer. The scheduler was easy to use, and I was able to quickly set up a series of scans for my PC.

It should also be noted that anti-virus technology is best paired with a good firewall for optimum security.

## Conclusion

ClamWin provided everything that would be required by most anti-virus users and then some; the lack of active scanning being the only drawback. It's a good replacement to the proprietary software you might be using now.

I was pleasantly surprised when I got a note about a new free software alternative to these costly proprietary anti-virus programs, ClamWin Free Antivirus (`http://clamwin.com`)

ClamWin also has an excellent community, with a good set of forums and very active developers. My queries were answered in only a few short hours, making technical support easy for non-technical users.

Organizations can look to ClamWin as a sign of things to come.

I hope this has inspired you to try out ClamWin for yourself, and perhaps to think twice when that "subscription fee" for your other scanner comes due. Or, if you've never used a

| ClamWin | |
|---|---|
| Name | ClamWin |
| Maintainer(s) | Alch |
| License | GPL |
| Platforms | Win2K, WinXP, Microsoft Windows Server 2003, Win98, WinME |
| **MARKS (out of 10)** | |
| Installation | 8 |
| Vitality | 8 |
| Stability | 9 |
| Usability | 7 |
| Features | 7 |
| **Final mark** | **8** |

virus scanner, hopefully this will give you new power to secure your digital life.

My queries were answered in only a few short hours, making technical support easy for non-technical users



## About the author


Robin Monks has been a technical writer for 2 years, and is actively involved with Mozilla, Drupal and various other free software projects.

Longhorn application development using XAML - learn tomorrow's GUI standard today

# Working together and sharing code with TLA

## TLA as your Version Control System

Carlo Contavalli

If you ever worked on a free software project or if you have ever worked as a developer, you probably know that managing source code, patches, and software release cycles is not the easiest task to perform. Things get even worse if lots of people are working on the same project: more code to manage, more people to coordinate, more patches to integrate and mainstream. Even if you don't write software or have never worked on such projects, I'm sure that as an addicted computer user sometimes you felt like "hey, why didn't I make a backup of that document", or "hell... I liked it better before" or "wow... why was that change introduced? I can't remember it...". In this article, I won't talk about the importance of backups, which should already be clear in every reader's mind, but I will introduce a new and very powerful system useful for managing source trees, tracking down changes, and allowing many people or groups to work together or independently on the same projects: TLA, often referred as GNU Arch, a new generation Version Control System (VCS).

### Version Control Systems and your happiness

If you don't know about VCS's or SCM's (Software Configuration Managers), just close your eyes (maybe at the end of the paragraph) and start imaging a big software project: lots of developers, lots of source files, roadmaps to inspect closely, bugs to find and fix, patches to be released while new features are being introduced, developed and tested. And yes, all these things being done by different people, for different reasons as time goes on.

As soon as a project starts to grow and spread among your customers or users, it becomes very important to have the ability to track:

- When and why a particular change was introduced?
- Which version of the software each customer/user has installed?
- Which changes were introduced in each version?
- Who it was that actually introduced a given change (often, a given bug)?
- Where a version of the software, which is still in use and which needs a minor change, can be found?

This is where Version Control Systems come to rescue. The purpose of a VCS is to prevent a given project, whether it be a software project or anything else that can be saved as a tree of files on your hard disk, from becoming a nightmare. Once in use, it will take care of:

- Remembering which files a project or a particular version of your project is made of
- Remembering changes that made to any file, allowing you to "undo" those changes and to know who introduced them and why
- Conflicts – preventing different people introducing "incompatible" changes to the same files without knowing
- Introducing automatisms into your software release cycles, by providing "hooks"

- Allowing you to continuously develop two or more versions of a project (often called branches), easing the task of exchanging and sharing code among them
- Easing the task of integrating changes (patches) provided by third parties

Generally speaking, a VCS could be considered some sort of oracle which knows everything about each and every file of your project, which you can ask questions or delegate tasks to (like, "please, apply this patch to the whole project... or please, show me that file as it was 20 days ago"). Once you start using VCS's you will become addicted, and will start using them for everything, even for simple documents or articles (like the article you are reading, and no, I'm not one of the most addicted).

A VCS could be considered some sort of oracle which knows everything about each and every file of your project

## TLA as your Version Control System

By now, you should be wondering why I felt such a strong need to talk about TLA in this article (no, the "Free Software Magazine" editor didn't force me, and no, the upstream authors didn't pay me), when so many VCS's are available on the internet.

First of all, TLA is one of the few free (as in free speech) VCS's available that comes with a decentralized development model in mind: it supports multiple archives, possibly managed by different people, created for the same project. It also provides support for archives to import/export patches and pieces of code from each other, without loosing track of who made them and where they came from.

Second, TLA is very powerful in what it does, still being extremely simple, both in its usage and in the way it works. For example:

- An archive is a simple hierarchy of directories, each one containing a .tar.gz with either the sources of the project or some sort of "diff" file and a couple of text files used internally by TLA (usually, the log entry and the checksums of the files). Other VCS's use Berkeley DB databases, SQL backends, proprietary database formats, etc.
- It doesn't need a dedicated server. TLA can store, access and write archives using standard protocols like SFTP, FTP or HTTP (using WebDAV for write support).
- TLA supports symlinks, renaming both directories and files, keeps track of privileges and file system permissions
- It supports GPG signed archives and operations, and more.

Choosing the right VCS is a matter of looking at the development model being used. Obviously, TLA doesn't fit well with every development model. While there are limits that only TLA is able to overcome and models that only TLA fits, there are other VCS's able to overcome some of TLA limits and that fit better with other development models. Be careful before taking any decision in any direction.

Choosing the right VCS is a matter of looking at the development model being used

## Starting up with TLA

Without going into the details of the various commands, details you can find in any of the many TLA tutorials, let's see what you need to do and how things work with TLA.
The first time you install TLA, you will need:

- To introduce yourself to TLA, by stating your name, email address, and the address that will be used by TLA to generate the changelogs
- To create your own archive or repository, a place where TLA will store all of your projects, or trees, and all of its own data
- To import one of your projects into TLA, and start working on it

Well, to introduce yourself to TLA, there's not much to say, just run something like:

```
$ tla my-id \
 'Carlo Contavalli <ccontavalli@masobit.net>'
```

Version Control Systems and the Linux Kernel

Almost every project above a certain size is managed using a Version Control System (often called a Software Configuration Manager, or SCM, even if the two terms have slightly different meanings). One such project, is the Linux Kernel.
As many of you probably know, the sources of the Linux Kernel are managed using a proprietary and closed source product, BitKeeper (http://www.bitkeeper.com).
Although this product has proven its reliability and usefulness over the course of several years, many have pointed out that its license, usage conditions and the fact it is closed source makes it incompatible with the development model behind Linux and the philosophy behind free software. These people suggest that the Linux Kernel development project should switch to using a Version Control System completely based on free software, and not rely on the free (as in free beer) availability of a closed source proprietary product.
This issue has been discussed in many occasions (flame wars?) on different mailing lists in the past. Linus Torvalds has explained on the Linux Kernel Mailing List (http://www.tux.org/lkml/) in 2002:

> "Would I prefer to use a tool that didn't have any restrictions on it for kernel maintenance? Yes. But since no such tool exists, and since I'm personally not very interested in writing one, and since I don't have any hangups about using the right tool for the job, I use BitKeeper... "

His position has probably given an additional burst to the development of new, more powerful, and free (as in free speech) VCS's, although the Linux Kernel sources are still (at time of writing) managed using BitKeeper.
On April 6, 2005, just a couple weeks ago, Linus Torvalds finally announced that due to a conflict over BitKeeper usage, the kernel team was looking for alternatives.
At the time this article was written, no choice was yet made upon which SCM to use, but candidates discussed on the mailing list were mainly monotone (http://www.venge.net/monotone/) and bazaar-ng (http://www.bazaar-ng.org/), based on the GNU Arch system.

Creating a new archive is a bit more work. You need to choose a name for it, some sort of label and decide where to store it. Choosing the name for your archive is probably the most important step: as each archive can be mirrored in different locations, or accessed using different methods at the same time (like HTTP, FTP, etc.), its name is the only piece of data that allows TLA to distinguish one archive from one another. It is also very important for any two different archives to have different names: a distributed VCS is able to work on code and trees kept in different archives, and if the archives involved in a project happen to have the same name, big problems can occur.

To avoid this kind of trouble, TLA enforces some policies over archive naming: an archive name must be made of an email address, followed by a "–" and by an arbitrary name, chosen by the user. As an example, a valid archive name could be: "ccontavalli@masobit.net–public".

As you may imagine from the previous paragraphs, the email address is there to guarantee some sort of uniqueness in archive naming. Without it, there would probably be thousands of archives named just "public" or "private", and TLA users would have a lot of trouble dealing with them. Obviously, you should use one of the email addresses you are the owner of, and if you want to keep using TLA without difficulty, you shouldn't have two archives with the same name.

This and many other conventions enforced by TLA have been the origin of many a flame war. If you don't like putting your email address in the name of an archive, just create a mailbox for it, or, like I've done many times for public projects, point it to a mailing list. After all, it's just a label.

Before talking about how to actually create an archive, there is another convention that TLA gurus suggest to use in nam-

ing archives: they suggest to put some sort of number in the archive name, like the year of when the archive was created. Archives tend to become pretty large, even if a very efficient storage method is being used. By putting some sort of number in the archive name, you have the freedom to close or "seal" that archive up when it starts to become too big, and to create a new archive with the same name (but a different number) where the development can easily continue.

Now that you know everything about archive naming, just create one, using something like:

```
$ tla make-archive \
 ccontavalli@masobit.net--2004-public /home/tla
```

Where `/home/tla` is the directory where you want your archive to be stored. If you wanted to create it on a remote server, you could have used something like:

```
$ tla make-archive \
 ccontavalli@masobit.net--2004-public \
 sftp://ccontavalli@a.server.net/home/tla
```

Where "sftp://" could be replaced by "ftp://" or "http://", with no need to have shell access to the remote end.

Always keep in mind that anywhere you keep your archives, privileges must be enforced by the underlying method: the file system, the FTP server or the web server. Make sure that you setup privileges correctly before opening up a TLA archive.

Now that you have your own archive, you can import any project you may want into TLA (or start a new one). Again, the hardest part of the operation is choosing a proper name for the project.

This time, TLA requires you to assign a name made of three parts, divided by "–", like "coolplayer–main–0.0". In this example:

- Coolplayer is the name of the project, it can be almost everything you like, with some restrictions on the characters being used.
- Main is the line of development for coolplayer.
- 0.0 is the version of coolplayer being kept in that archive.

There's not much to say regarding the name of the project: just choose one. It's a bit harder to talk about the name of the development line and the version number, so I'll start with a simple example.

Let's say you're one of the "coolplayer" developers, and the currently released version of coolplayer is "0.0.2". As you work, you're probably adding features to version "0.0.2", and preparing to release version "0.0.3". So, you are working on improving version "0.0". At some time you may start introducing big structural changes, getting ready to release version "0.1.x". If you work on those changes directly into the "0.0" branch, you would probably end up in making the code unusable until your work is completed, blocking bug corrections and all the other developers still working on "0.0". In this case, a good idea would be to create a project called "coolplayer–main–0.1", starting from a particular version of "coolplayer–main–0.0", and adding all of the changes planned for version "0.1" there, still allowing "0.0" to be developed and released without changing the release procedures.

TLA would allow the two branches to import/export patches from each other, keeping track of what has already been done in each of them.

Now, let's say your company is selling "coolplayer" to some of its customers and one these customers wants to buy version 0.0 but with some slight changes. In this case, you could just add those changes to the main version of "coolplayer", but perhaps you don't want to or you are not allowed to. A better solution would be to create a new line of development called "coolplayer–customer1–0.0", derived from "coolplayer–main–0.0", then the changes needed can be introduced while still keeping your boss happy. If a bug fix is introduced in "coolplayer–main–0.0", you can use TLA features to import the bug correction into the line "coolplayer–customer1–0.0", without losing any of your previous changes. TLA will handle this.

By now, you might be wondering what the "main" stands for. You're free to follow any convention you may like; just be sure you do have a convention. Usually any given software project has one "main" line of development which is used to add new features and to prepare for new releases.

In TLA, branches are natural, there's no magic behind them

On other VCS systems, lines of development are called "branches". In TLA, branches are natural, there's no magic behind them. The assumption is that each and every project

will always have multiple lines of development, and could be "derived" from previous projects. TLA always forces you to give it a name.

There is one more thing to be said about branch names: once a version of "coolplayer" is released, we need to tell TLA "here in the archive, you have a version of coolplayer that I want to remember, please save it into... " - where that "into" is usually the name of another archive. So, for coolplayer 0.0, you may end up having the following branches:

- coolplayer–main–0.0: where all the development is made
- coolplayer–release–0.0: where all the coolplayer public releases are stored (the "into" mentioned above)
- coolplayer–customer1–0.0: where the coolplayer with changes for customer1 are kept

On other VCS systems, this operation is called "tagging". On TLA, tagging just means "store this version of the current project in some other branch", which is by all means identical to any other branch. You can then work on any of the above branches as you wish, and easily import/export changes from one branch to another without issue, using standard TLA features.

Now that we've chosen the name for your "branch", or project, let's create it:

```
$ tla archive-setup \
 -A ccontavalli@masobit.net--2004-public \
 coolplayer--main--0.0
```

There's one small problem: the branch is completely empty, no files have been put into the archive, so you still can't do much.

To start using TLA, just go into the directory where you started working on your project (or an empty directory), make a backup (just to be safe), and run:

```
$ tla init-tree \
 -A ccontavalli@masobit.net--2004-public \
 coolplayer--main--0.0
```

This tells TLA the directory contains all of the sources for the coolplayer project, which needs to be stored in the 2004-public archive. Don't get scared by '{arch}' or '.arch-ids' directory that TLA will create in your source tree. You now need to tell TLA which files and directory you actually want it to keep track of, and which files you want to keep in the TLA archive.

To do so, just run:

```
$ tla add file_or_directory_name
```

After you have added all of the files, you can run:

```
$ tla tree-lint
```

to verify you haven't forgotten anything. Note that tree-lint will output warnings regarding files TLA knows nothing about, and errors regarding files which violate "naming conventions". You can slightly tune "naming conventions" by editing the configuration file '{arch}/=tagging-method', created by TLA in your project directory.

As with any VCS, it is a little tedious keeping it informed about which files are part of the project, considering that every rename or remove should be reported using some sort of command (with TLA, tla rm, tla mv, etc.).

Thankfully, TLA utilities comes to the rescue again: you can easily find scripts like "tla-update-ids", which tells TLA about any new files, and files which have been removed, etc. by reading the output of "tla tree-lint".

Finally, you can tell TLA to upload your project into the archive:

```
$ tla import
```

## Working with TLA

You can finally start working on your project as you wish. You have it on your hard drive, so you can just work as usual by editing your local copy of the files.

When you are done, and you have a set of changes you want to be saved on the archive and make them available to other developers, just run:

```
$ tla commit
```

If you set the "EDITOR" environment variable up correctly, your editor will pop up asking you for an entry to be added to the TLA log; this entry should list the reason for the changes.

Otherwise, you will have to edit the file named "++something" in your source tree manually and then run "tla commit" again.

While committing, if some other developer made changes to the project, you will receive a warning and the commit will be automatically aborted. You will then have to manually choose what to do. Probably, you would run:

TLA and naming conventions

TLA is quite a nice tool. However users often feel that the naming conventions enforced by it are tedious and wrong.
Directory or file names which contain characters like "{", ",", "++", "=" are used by default by TLA utilities, and are sometimes hard to handle.
Commands like "archive-setup" or "make-archive" that perform completely different tasks on "objects" they both call "archives" are not very easy to use.
TLA is evolving, and the naming conventions are being changed for the better as time goes by. As an example, in version 1.3 a "delete-id" operation has been added as an alias for "delete", which will probably disappear in a couple versions.
Other users feel that TLA is too restrictive in naming archives or projects. You may get frustrated with this in the beginning, but after a while you'll probably come to like it, and start feeling that this strictness is necessary.
If you like TLA but not its commands nor its naming conventions, you may want to try "bazaar", a fork of TLA, with cleaner and neater commands and naming conventions.

```
$ tla update
```

to update your local copy of the tree with the changes introduced by the other developers, without loosing any of your own local changes.

If something goes wrong and one of your local changes can't be merged automatically with those introduced by other developers, a .rej file is created, containing the code that caused the problem, exactly as the "diff" and "patch" command would (guess who created that file?).

If you'd ever like to have another copy of the project on your computer or any other computer, you need to perform two steps:

- tell TLA how the archive, as you named it, can be reached
- get a copy of the branch you are interested in

As an example, if I wanted the sources of "coolplayer–main–0.1", I'd need to run something like:

```
$ tla register-archive \
 ccontavalli@masobit.net--2004-public \
   sftp://ccontavalli@a.server.net/home/tla
```

to tell TLA that the archive named "ccontavalli@masobit.net–2004-public" is accessible from my machine by connecting with the SFTP protocol to a.server.net, and:

```
$ tla get -A \
 ccontavalli@masobit.net--2004-public \
 coolplayer--main--0.0
```

to actually get the project into the current working directory. Take note that "tla get" as executed above will fetch the latest version of the project. If you want to fetch a previous version, you just need to know that every commit into a tree has a name, usually "patch-x", where "x" is incremented on each commit, while each import operation has a name like "base-x". So, if you want to fetch your sources as they where imported in your TLA archive, you just need to run something like:

```
$ tla get -A \
 ccontavalli@masobit.net--2004-public \
 coolplayer--main--0.0--base-0
```

As with any VCS system, you also have lot of commands that allow you to browse the content of a given archive or the state of a project. TLA provides for example:

- logs - to have a list of commits performed on a given project, possibly with a small summary of why they were performed (–summary)
- abrowse - to see the list of the projects in a given archive
- missing - to have a list of missing "commits" on your current tree
- and many, many others

Up to now, I have talked about "tagging" but haven't shown how it's done. Suppose that a particular version of one of your projects is ready to be released and you want TLA to remember that version as one of the released versions. To do so, you can quite easily ask TLA to save it in a different branch, for example, in the "release" branch, just to choose one:

```
$ tla tag -A \
 ccontavalli@masobit.net--2004-public \
 coolplayer--main--0.0 \
 coolplayer--release--0.0
```

This way, the latest version of coolplayer in the main branch for version 0.0 is saved in the release branch.

So far, I've only shown TLA being used as I would have used any other centralized VCS system: one repository, several lines of development, developers that get and commit from that single archive.

The "tag" command is at the base of distributed development, since it allows a particular version of a given project to be saved in any other archive, while keeping track of its origin and allowing others to work on that archive:

```
$ tla tag \
  c@m.net--2004-public/coolplayer--main--0.0 \
  s@a.net--private/coolplayer--myown--0.0
```

[in the above example, `ccontavalli@masobit.net` and `somebodyelse@another.net` had to be changed in `c@m.net` and `s@a.net` for typographic needs. Please do not take this as an excuse not to use tla: even if its command lines may sometime get lengthy, tla is extremely useful and powerful and a properly configured environment can be of much help.] After executing the above command, somebodyelse@another.com could work on coolplayer from their own archive. If something good comes out, both projects can fetch changes from each other by using commands like "tla star-merge", "tla replay" and so on, commands that are beyond the scope of this article.

## Conclusion

TLA is a complete and extremely powerful VCS. There are lot of utilities available, starting from web interfaces, up to automatic update tools (like tla-update-ids).

TLA also provides many extremely interesting features that were not even introduced in this article, like transparent support for archive mirroring, GPG signed archive support, client-side hooks and advanced merging features.

For those concerned by the long command lines being used, a properly configured environment can be of great help, and a lot more can be taken out of TLA:

- "tla my-default-archive" allows you to tell TLA which archive to use if no -A is used.
- Bash completion for TLA archive names is provided by external scripts allows you to use TAB-TAB to complete your command lines.
- "tla make-log" allows you to write your changelog while working, without having to remember all changes until commit time.
- And there's a lot that can be said about multi-tree projects, revision libraries, and changesets.

The main shortcoming in TLA that I have encountered so far is caused by the lack of server-side hooks. As TLA uses standard protocols, it's not possible to tell TLA to run a given command on the server every time a commit is performed. Also, if a client crashes due to networking problems during an operation on the archive, the archive itself may be left in an undefined state.

Luckily, the simple storage used by TLA can be manually fixed in just a few minutes, and dnotify or inotify support in the Linux Kernel allows simple hooks to be called on the server when needed, even if no direct support is provided by TLA. And, if you really want to, you can always try "arch-pqm".

## Copyright information



## About the author

Carlo Contavalli works as a full time developer for Masobit Corp. (http://www.masobit.net). He also is an addicted free software activist. He has contributed to projects like PigeonAir (http://www.pigeonair.net), mod-xslt (http://www.mod-xslt2.com), Debian GNU (http://www.debian.org), and many others, by writing code, submitting patches and writing documentation. He has also organized or taken part in various free software related events.

# Initialization sequence in GNU/Linux

## The process of booting your PC, from power to prompt

Steven Goodwin

The login prompt is a nice place to be. Poised, fingers on keyboards, ready to send mail, surf the web, or do a little programming. However, from power-on to login prompt there is a long road for our GNU-powered friend to travel.

### The boot up

The first step on this journey is the BIOS power on, system test. Or POST for short. BIOS stands for Basic Input Output System, and is the custom chip on the motherboard that holds a small program to kickstart the whole boot-up. Its job is to check the low-level hardware components, such as memory and hard disks. Configuration of the BIOS can be a complex task, and is certainly an article in itself. If you're curious, hold down the delete key during boot-up to see the BIOS configuration screen. Control is passed from here to the boot loader.

The boot loader is a very small program that lives on the hard drive. It is not stored as a file, since that requires a file system—and one doesn't exist yet since we haven't loaded an operating system. So instead, it is stored in the very first 512 bytes of the disk (also known as the master boot record, or MBR). The BIOS only needs to know the physical layout of the hard drive (which it obtains from the POST) to load this program into memory, and execute it.

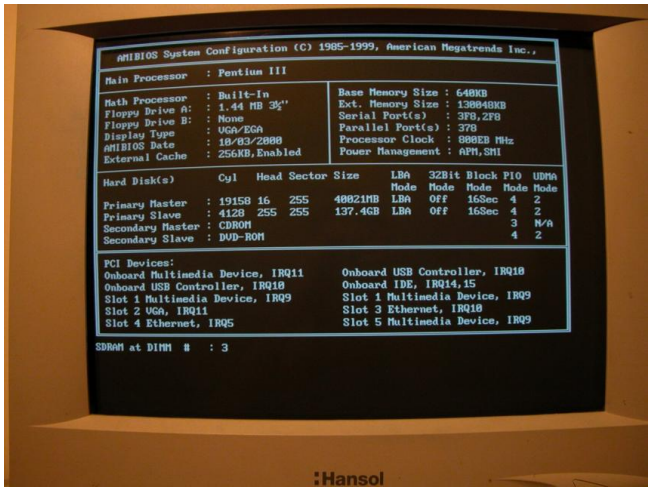
The whole boot process follows this pattern of small incremental steps; each doing a specific job, and passing control to the next program. The term "boot" stems from the phrase "bootstrapping", meaning to lift yourself up by your bootstraps.

The beginning of the POST


> The term "boot" stems from the phrase "bootstrapping", meaning to lift yourself up by your bootstraps

The boot loader program itself is operating system-independent. Indeed, one boot loader can provide you with access to several OSes, often through a basic menu screen, facilitating a dual-boot machine. Most GNU/Linux distri-

Typical configuration data discovered by the BIOS




butions ship with a choice of two boot loaders, Lilo and GRUB. Lilo, for example, uses `/etc/lilo.conf` to determine which kernel, or kernels, can be loaded at boot time. This file can also be used to pass options from Lilo into the kernel. In order to create or change the Lilo boot loader you need to edit `/etc/lilo.conf` and run the `/sbin/lilo` program. This takes the loader and places it directly into the MBR. The most common change of this kind occurs after compiling a new kernel, as the boot loader needs to know where the new kernel lives. Since 512 bytes doesn't provide much room for code, most boot loaders are (transparently) loaded in two parts. The next evident activity comes from the kernel, in this case, Linux.

Linux, regardless of version, will always issue a flood of messages to the screen as it runs. These describe the computer system, as the kernel sees it, to aid troubleshooting. The messages themselves are also written into a log file at `/var/log/kern.log`, or can be retrieved using the `dmesg` command.

The kernel doesn't have any programs to manage yet, so it does little more than initialize itself. This includes preparation of the all-important file system. Most file systems will be built into the kernel

directly, but for some drivers (such as SCSI) these will be loaded into the RAM disk.

The RAM disk (or `initdisk` or `initrd`) is a memory-based file system that is created by the boot loader before Linux is run. A more exotic system configuration can be created using the `/linuxrc` file to load other drivers, or mount additional components into the file system. Linuxrc may be either a program, or a script. However, the latter would require a shell interpreter in order to run, and is thus rarer.

The Linux kernel, as its final act during boot-up, runs `init`. This is a program normally stored as `/sbin/init`.

## Alternate locations for `init`

By default, `init` will live in `/sbin/init`, and is the first place the kernel will look, followed by `/etc/init` and `/bin/init`, in that order. Should all of these fail, a shell (`/bin/sh`) will be executed instead, enabling you to recover the system. You can also specify a different location for `init` by passing it as an argument to the kernel. Lilo users, for example, can include an `append` line in `lilo.conf` to use a completely different program. This line can also specify a runlevel that will take precedence over the one in `inittab`.

```
append="init=/sbin/myinit 5"
```

It is `init`, and not the kernel, which then configures the system, starts services (a more correct term for applications such as Apache) and opens virtual terminals. And this is where the real fun begins!

## Once the kernel has started

`init` prepares the machine to work in a particular way: what software should be running, what terminals should be open, and which network services are to be allowed. It does this using *runlevels*. A runlevel describes the basic running state of the system, so naturally you can only be in one runlevel at a time.

A runlevel describes the basic running state of the system

There are a number of runlevels that are, by convention, used to describe a working Linux system, with 2 or 3 being usual for a multi-user configuration. The complete list is given below (according the Linux Standards Base - LSB).

- Runlevel 0 (halt) - Shuts down everything and brings the system to halt.
- Runlevel 1 (single user) - Useful for maintenance work.
- Runlevel 2 (multi-user) - multi-user, but with no network services exported.
- Runlevel 3 (multi-user) - normal/full multi-user mode.
- Runlevel 4 (multi-user) - reserved for local use. Usually the same as 3.
- Runlevel 5 (multi-user) - multiuser, but boots up into X Window, using `xdm`, or similar.
- Runlevel 6 (Reboot) - As 0, but reboots after closing everything down.

Most systems default to 2, 3 or 5.

Although the kernel starts `init` at the default runlevel, this can be changed at any time, provided you are the superuser, without rebooting your computer.

```
# init 2
```

This will switch the system into runlevel 2, starting all services that should be running at this runlevel, and killing those that shouldn't. Using runlevels as a profile in this manner lets you remove services during system maintenance, such as the network, very simply.

Looking at the runlevel table again you will notice runlevel zero is called "halt". This is not a misprint! Since a runlevel describes what services should be running, switching to a runlevel that closes all of its services and runs a program to halt the processor would be a considered shutdown procedure. Therefore:

```
# init 0
```

is equivalent to the more descriptive:

```
# shutdown -h now
```

Although longer, the latter is preferable because it is more extensible; letting you shutdown in half an hour, say.

## Table for one

As with most programs under Linux, `init` has its own configuration file stored in `/etc`. It is called `inittab` and consists of comments (beginning with the over-familiar # symbol) and configuration data that indicates what, where and when particular services are to be run. The `inittab` file itself is well commented, and worth reading.

The first part of the file tells us:

```
# The default runlevel.
id:2:initdefault:
```

This format is typical, as each line in `inittab` has four fields separated by a colon. The first portion is an *identifier* for the action, and can be anything provided it is unique within the file (with the exception of the virtual terminals, which will be covered later). The second indicates the runlevel(s) to which this rule applies. In the above case it means runlevel 2 only, while the case below would work in all "multi-user" levels.

```
message:2345:wait:echo "In multi-user mode"
```

Parameter three is called the *action*. It indicates how the command (given in parameter four) is to be run. There are numerous actions available, and most are used in the `inittab` provided by most default distributions of GNU/Linux. The common ones are shown in the list below.

- `wait` – Execute the command, and wait until it completes before moving onto the next one. Used mostly for running software in sequence. If a program can not be run, `init` will work through the rest of the file. `init` does not stop on errors, but will report them to the console.
- `respawn` – Execute the command, but respawn it when the process completes. Used for virtual terminals that need to re-run the login prompt (through `getty`) whenever the user logs out.
- `ctrlaltdel` – Whenever the "three fingered salute" is given, run this program. It is usually configured to reboot the machine.
- `off` – Disables the entry, without deleting it from the file.
- `initdefault` – The default runlevel used if `init` is called without an argument.
- `sysinit` – The command is run first before anything else, when the system boots (runlevels are ignored).

Finally, the executable in parameter four may be a command, script or daemon, and can include arguments if required. But both the command and its arguments may be omitted, as we saw with `initdefault`.

## The switch

`init` is a simple beast. It reads through each line in `inittab`, executing every command (relevant to the current runlevel) in the order in which it's presented in the file. The first command invariably performs system initialization, by specifying the sysinit action.

```
si::sysinit:/etc/init.d/rcS
```

This is the Linux equivalent to the `autoexec.bat`, or `confis.sys`, files from Windows and DOS. Its purpose is to configure any system-wide parameters (such as the system clock, or the serial port), regardless of runlevel. Once `init` has handled the system initialization it switches to the default runlevel and continues reading the rest of the file.

The sysinit start-up code is handled by the `/etc/init.d/rcS` script, which starts each process that is catalogued in the `/etc/rcS.d` directory. Since the boot-up sequence doesn't have a runlevel, a pseudo-runlevel called 'N' is used.

When switching between runlevels, the set of running services must also change. While this is possible to do from inside `inittab` and the `/etc` directory, it is cumbersome. To ease the pain, Linux uses a set of directories, one per runlevel (called `/etc/rc0.d`, `/etc/rc1.d`, `/etc/rc2.d` etc), that describe the services that must (and must not) be active in that specific runlevel. A script called `/etc/init.d/rc` is then responsible for starting and stopping them. The directory structure and absolute location of `rc0.d` and `init.d` can be inferred from the `inittab` file. If yours differs then refer to the SysVInit textbox in this article.

`init` is clever enough to start services in the correct order

Taking runlevel 2 as an example, this has a directory called `/etc/rc2.d` which contains a number of files. Those beginning with the letter 'S' are services that will start when we change into this runlevel, and those beginning with 'K' are services that will be killed.

`init` is clever enough to not stop, and then restart, any service that appears in the both the new runlevel, and the previous one. It is also clever enough to start them in the correct order—that order being one that you (the user) has numerically set up.

```
S10sysklogd
S12kerneld
S14ppp
S19bind
S19nfs-common
S20anacron
```

This fragment of the `/etc/rc2.d` directory from a GNU/Debian box shows that `sysklogd` will be started first, followed by `kerneld`, then `ppp`, and so on. The `/etc/init.d/rc` script will execute each of these files in order, passing it either the argument "start" or "stop", depending on whether the filename begins with an 'S' or 'K'. This gives the `/etc/rc2.d` directory similar functionality to the Windows Startup folder. But while Windows hides the order and method by which its startup program runs,

SysVInit

SysV Init has been incorporated by the Linux Standards Base as the method for system initialization. Depending on the distribution and version you use, the locations of certain files may be different from what we've given here. Firstly, the `init` program might live in `/sbin` and not `/etc` (although most distributions moved it there some time ago). And secondly, all of the runlevel configuration directories exist not in `/etc`, but in `/etc/rc.d`. The latter directory also holds the `rc` script, `init.d` directory and all other data mentioned above. Its working method, however, is no different. The `inittab` file will detail where your scripts live. Two scripts that also appear are `rc.local` and `rc.sysinit` (the latter will in turn usually run `rc.serial`). The execution order here is that `init` will run `rc.sysinit` first (configuring the network, checking the file system, and so on), followed by the runlevel scripts, and finally the `rc.local` file. If they don't exist, no error is produced, and Linux continues booting as normal.

Linux makes them available with ease, executing them directly from the `rc` script.

The `sysinit` directory `/etc/rcS.d` works in exactly the same way, but as it contains only system configuration information, no 'K' files are required. In contrast, booting up into single-user mode (runlevel 1) has almost nothing except kill files, to stop all the old services.

## The missing link

The files in the `/etc/rc2.d` directory, however, are not actually files. They are links to scripts that do the real work! The script for `S10sysklogd`, for instance, would reside as `/etc/init.d/sysklogd` (the `S10` having been stripped off) and would start or stop the service based on its argument.

All the startup and shutdown scripts are in this directory (`/etc/init.d`), so controlling services on-the-fly is very easy. You can stop, start, or restart them with one simple command. Namely,

```
# /etc/init.d/apache start
```

Controlling services on-the-fly is very easy. You can stop, start, or restart them with one simple command

Some scripts will also support the restart directive. This removes the obligation to kill each process, and restart them manually, whenever a change to the configuration file is made. It also removes the need to reboot after installing new software, since the start command can be called directly.

You *can* add these links yourself with the `ln` command:

```
# ln -s /etc/init.d/apache /etc/rc2.d/S20apache
```

However, adding the same link to several runlevels (and its equivalent kill version to the halt, shutdown and single-user runlevels) can be a little monotonous, and therefore prone to user error. So there is a script that helps. It's called `update-rc.d`, and has the usage:

```
usage: update-rc.d [-n] [-f] <basename> remove
 update-rc.d [-n] [-f] <basename> \
   defaults [NN | sNN kNN]
 update-rc.d [-n] [-f] <basename> \
   start|stop NN runlvl runlvl .
 -n: not really
 -f: force
```

The basename is the name of the script, and would be `apache` in our example.

```
# update-rc.d apache remove
```

This would remove all the Apache symlinks in the `/etc/rc?.d` directories (where "?" can be any character). You must make sure the `/etc/init.d/apache` script is also removed, either by manually deleting it, or by use of the `-f` flag.

The `defaults` option will start the service in all multi-user runlevels, and stop it in the single user, halt and reboot runlevels.

If you want to add services to specific runlevels, the "start|stop" option must be used. This requires the order parameter. The runlevels are given as separate parameters, and terminated with an all-important full stop.

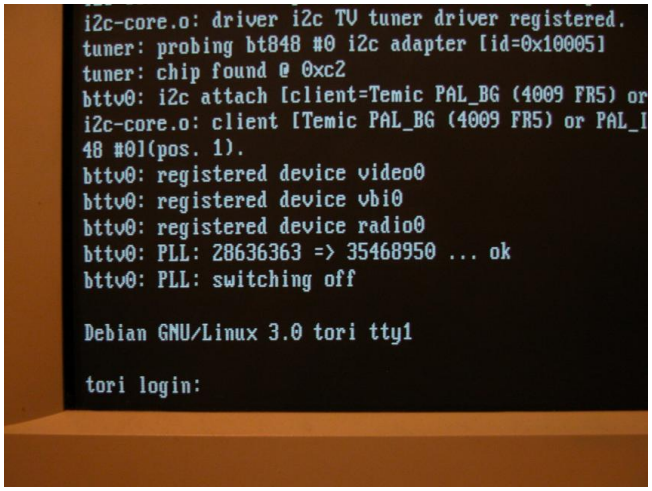
```
# update-rc.d apache start 20 3 4 5 .
 Adding system startup for /etc/init.d/apache \ldots{}
   /etc/rc3.d/S20apache -> ../init.d/apache
   /etc/rc4.d/S20apache -> ../init.d/apache
   /etc/rc5.d/S20apache -> ../init.d/apache
```

The NN symbol has been replaced with a number indicating the placement of the service in the start-up sequence. This represents the positional number we saw earlier on each of the symlinks, and can be omitted when using defaults. The default value is 20—a sensible choice since the network components are ready by this time.

## The end of innocence

Once the various services have started, the `inittab` script arrives at the actions to configure the three-finger salute (ctrl-alt-delete) and prepare the virtual terminals. These work simply by running the `/sbin/getty` program (or equivalent) on each specified terminal. The respawn action is required to repopulate the virtual terminals after a user has logged out. If the terminal is connected via modem, `inittab` can handle that too by using `mgetty` instead of `getty`. At this point we have a login prompt, and our journey is over.

Ready at the command prompt




## Conclusion

The apparently simple act of getting a prompt on-screen is very involved. However, it is not as complex as it first appears since it is comprised of several small steps, each building on the previous one.

Each step increases our understanding and helps us streamline our system.

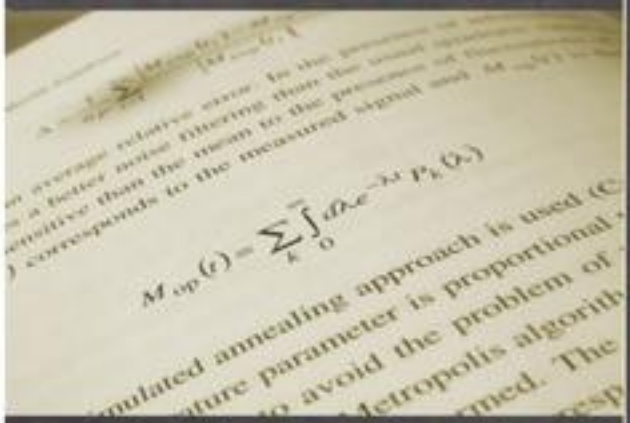
## Copyright information



## About the author

When builders go down the pub they talk about football. Presumably therefore, when footballers go down the pub they talk about builders! When Steven Goodwin goes down the pub he doesn't talk about football. Or builders. He talks about computers. Constantly...

**Communicating mathematics...**



**...is getting easier.**

# Math Focus

News
Discussion
Collaboration
Software
Books

www.mathforge.net

**Powerful tools for online mathematics**

mathforge

# Disaster relief and free software

## Commercial software distribution and development in perspective

Aaron E. Klemm

Technical needs in the immediate aftermath of the South Asian tsunami disaster of 2004 put software development and distribution methods into sharp focus for relief groups. When volunteers were immediately available to help coordinate relief efforts, access to software slowed them down. It was evident that traditional commercial software distribution had broken down. It was untenable.

The free software distribution model, on the other hand, requires absolutely no modification to make tools available when people need them. Free software tools such as available in a typical GNU/Linux distribution do not have to suspend their normal distribution and development methodologies to help volunteers who need software immediately.

In Sri Lanka after the tsunami, one blogger in particular, along with volunteers from several government agencies and software advocacy groups, worked through creating a new project to coordinate relief efforts. Along the way, a series of lessons for free software proponents became obvious. Not obvious in their novelty, but obvious because of the immediacy of such a disaster. Standard business procedure was made to look silly in the face of the simple need for a software tool to organize relief requests and resources.

### Lessons and roadblocks

The blogger was Sanjiva Weerawarana who took the lead just days after the tsunami to bring together software developers in coordination with Sri Lankan government groups to create Sahana, a software package designed to provide a comprehensive and extensible disaster relief system released under the GNU General Public License (GPL).

The 2004 tsunami. Photo by Sofwathulla Mohamed

The Center for National Operations in Sri Lanka was set up to coordinate relief efforts for the country. They were immediately overwhelmed with relief requests and did not have even a simple system to organize the information coming in.

Weerawarana, a free software advocate in Sri Lanka and a recently elected member of the Open Source Initiative's board of directors, was in a unique position to help launch a disaster relief software effort that could become a global

Free software

resource for similar future disasters. Sahana's primary function is to coordinate relief needs with relief resources across the many organizations providing aid to tsunami victims.

The implied lessons in the tsunami stories below are both philosophical and practical. To begin with, Richard Stallman's moralistic idea that free software "respects the freedom of the person using it" is easier to understand in this case than it ever can be inside the IT department of an average business. The moralistic side of the argument may never take hold in business, but in disaster relief it is obvious and can be the starting point for pragmatically opening people to the ideals of free software.

A practical lesson is that building a coordinated disaster relief software system requires cooperation between both businesses and governments in order to be truly helpful to disaster victims. Free software has been instrumental in forcing businesses to adhere to standards. Think of HTTP and the Apache web server as a prime example. Free software may be able to do something similar to encourage governments to share software development resources and disaster data.

In a specific incident of trying to acquire Microsoft Windows licenses to use on donated computers, Weerawarana and his colleagues learned that user's unfamiliarity with some free software products is still a severe impediment to its uptake. In this case, even in a time a crisis, the volunteer's comfort with Windows trumped the sanity of simply downloading a fully-functional GNU/Linux distribution.

> "Unfortunately in the meantime [a] couple of other folks contacted the local [Microsoft] people and convinced them that they had a PR disaster in their hand if these licenses were not given," Weerawarana wrote.

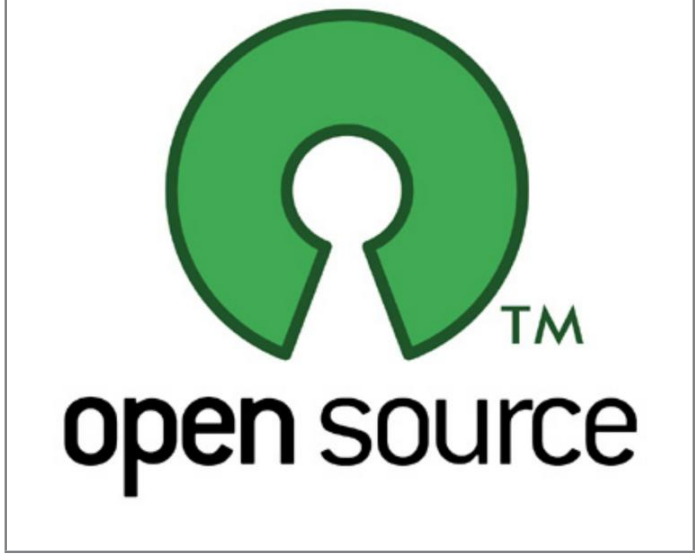
Finally, if Sahana is a demonstrable success, it shows a clear path for volunteers to propagate free software deployments in disaster relief functions. The Sahana story shows how free software is good for society and provides governments an efficient path to acquire useful disaster relief software. If all of this is true, the benefits easily translate to commercial software distribution.

## A licensing debacle

Weerawarana blogged in detail about his software efforts following the 2004 tsunami. One of the first conflicts he encountered was obtaining licenses for donated hardware.

According to Weerawarana, IBM donated 15 laptops to LSF, but Microsoft failed to provide free licenses for Windows XP. The hardware showed up with PC-DOS installed

Open source



instead which was not going to be terribly useful to the volunteers who were ready to use the equipment.

The laptops were used for gathering data for Sahana. Volunteers took the laptops into affected areas to gather data about affected people, damage reports, and relief resources.

> "Well they again refused a few days ago and I had just sent a note to the local LUG guys to come and install a Linux distro which we can give to normal people (maybe with a bit of help)."

Anyone who has struggled with licensing rights on their home computer with Windows XP and contrasted it with the ease and encouragement GNU/Linux distributions offer—to simply download, install, and go without worry—will shake their head at this story.

## Exposure makes the philosophy apparent

In the end, free software did not win the day on these donated laptops. Microsoft relented and the LSF used Windows XP because the users were more immediately familiar with it. It is difficult to imagine this would have been the case if Apple computers, for example, had been donated. As GNU/Linux desktops gain exposure, this type of situation will be less likely.

Imagine if these fifteen laptops were handed to GNU/Linux users. They would have immediately downloaded their preferred distribution and began working without any thought of securing a license—that concept is not part of using free software. No time would have been lost to bureaucracy, legal issues, and public relations.

Now imagine the volunteers were just as comfortable with GNU/Linux desktops as any other—a situation we can expect in the future—they may have paused for a moment to consider how ridiculous it is to wait for a software license before starting their critical work.

When their lives returned to normal and they were back at their day jobs, perhaps the inefficiency of waiting for a license at work would be more apparent as well. In a time of emergency, software users see that commercial distribution is a contrived process designed to protect the "owners" of code.

Weerawarana is a founding member of the Lanka Software Foundation (http://www.opensource.lk)


## Get free software in the door

A common story about the usage of free software in businesses is that it simply finds its way onto desktops and servers. IT staff need quick fixes and they know nothing is quicker than grabbing free software solutions. Users may find that Firefox treats them better at home so they install it at work and so on.

While it would be nice to see more top-down decisions being made in favor of free software, this under-the-door phenomenon has been working well enough. In fact, once free software is discovered to be working so well it will be hard to justify removing it. It proves itself in action.

The same thing can happen in disaster relief. Programmers looking for useful areas to donate their time can focus at the community level. Solving problems that affect people's lives will catch the eye of community and emergency response organizations.

Each time a group can solve civic problems like the Sahana volunteers have done, they will have demonstrated the viability of free software and slipped it in the door just as happens in IT departments.

The difference, of course, is that solving a civic problem has a greater impact than simply solving a technical problem in a server room. Leveraging these opportunities could be a boon for free software.

## Facilitating cooperation

In the most abstract sense, Sahana is designed to collect and manage data. Commercial data management systems often come crippled with a proprietary storage mechanism (or even license agreements that restrict how convenient the product is) so that while the software manages the data well, the vendor manages to determine how the data can be accessed.

The result is that such systems are sold with the expectation that an entire group of people (a company) will use

the same server (Microsoft Exchange, for example) and the same client (Outlook, for example). When a company decides to use a hegemonic system like this, they know (one would hope) that they are committing to it exclusively.

Unfortunately, the requirement that entire groups of people adhere to one server and one client fails when cooperation is needed across several businesses, governments, and relief agencies that are trying to coordinate relief efforts.

Sahana provides an open way to input and retrieve victim data. It's open because the source code is available. Commercial companies could not open up their product to allow multiple client access even if it would help relief efforts. They are selling a crippled data model that only works inside a single business and offers no technical advantages.

Sahana is a web-based system for entering and retrieving the relief effort data. As information is entered into the system by Sri Lankan volunteers about Sri Lankan issues, that information can be available on the web. Neighboring countries are able to access, cross-reference, and add pertinent data for overlapping efforts.

## A simple government demand can ensure business resources

The Sahana story showed how free software can be immediately useful. The fact that any volunteer with the right skill can jump in and start working on a project makes the free software process nimble enough to be the most efficient method.

If Sahana increases government interest in free software and programmers continue to aim their efforts at these critical areas, governments can be encouraged to demand open development of disaster relief software.

Rather than take what a business offers—say a proprietary information management system—and work as best as they can with it, governments will have an easier option. They will be able to ask businesses for time and expertise rather than restrictive software tools whose usefulness is questionable at best.

Businesses that truly want to help a technology effort following a disaster can assign programmers to open development tasks. Rather than give a resource that is partially useful, businesses have an opportunity to provide 100% useful resources to those in need.

Weerawarana contacted UN workers

## Conclusion

When computer users discover free software, it is rarely under such dire circumstances as a tsunami. Even so, the software demonstrates the principles under which it is developed, by being stable, useful, and configurable. In crisis situations, the argument for free software is self-evident. The Sahana story is one of better aid response for victims and opportunities for free software advocates to demonstrate a better way to distribute and develop software.

## Resources

Sanjiva Weerawarana's blog (http://www.bloglines.com/blog/sanjiva)

Sahana project page (http://sahana.sourceforge.net/)

## Copyright information



## About the author

Aaron E. Klemm is a free software advocate and the co-founder of Mathforge.net. He lives and works in Seattle.

# Towards a free matter economy (Part 2)

## The passing of the shade tree mechanic

Terry Hancock

> Of course, the construction of a free road does cost money, which the public must somehow pay. However, this does not imply the inevitability of toll booths. We who must in either case pay will get more value for our money by buying a free road.—Richard Stallman

Past economic models relied much less on proprietary manufacturing and consumer product sales, with more emphasis on manufacturing and repair as services called on by users who considered self-repair and self-manufacture as defaults. Despite modern concerns to the contrary, this approach is not destructive to a commercial free market. If anything, such an economy represents a more ideal capitalist marketplace. The creation of a substantial free-licensed development bazaar will provide an environment in which companies catering to these needs can thrive and become much more competitive with the corporate consumer product manufacturers which presently dominate our economics. Clearly this is a necessary shift if we are to obtain the flexibility and self-sufficiency needed for developing the new frontier in space.

As we consider the needs of new pioneers, it makes sense to draw on the experiences of old ones. During the westward migration of the nineteenth century, American settlers travelled mainly by "covered wagon"—a rugged cart made of wood, with some iron or steel parts, and a canvas covering to protect the occupants, drawn by oxen or horses. There were no "brands" of covered wagons, although there were several common types and sizes. Nor were wagons produced in factories. Instead, they were usually made by the settlers themselves, with the help of manufacturing services from local blacksmith shops. Repairs were handled in the same fashion[1].

A wagon is transparently reverse-engineerable, there is very little of its mechanism that is either invisible or non-obvious in its function, at least once you've seen it work. So naturally, anyone with some mechanical skill could make parts or whole wagons on their own. Patents were much harder to get back then, and in any case, the construction of a wagon was well-understood and therefore unpatentable. Even if a particular component was patented, this was easy to avoid, since with so few parts, there were relatively few interfaces to constrain designs and alternate methods could be used to do the same job.

Even as late as the 1950s, when the automobile had long since replaced any form of horse-drawn vehicle, and corporations had taken over the manufacturing role, Americans took pride in learning to repair and customize their own cars. For about four generations, America was graced with a new class of skilled amateurs known as "shade tree mechanics".

Engines of that era were still almost exclusively mechanical devices. You could get repair manuals, which were useful, but to a skilled amateur or professional mechanic, the important aspects of an engine could be easily figured out by disassembling it. Some engine parts were patented, of course, but not as many as you might think. Any patents on

Figure 1a: Covered wagons were built by settlers themselves or contracted from local builders, using blacksmith shops to make the more difficult parts. Enough information to service just about all wagons could be contained in one book, such as **Practical Carriage Building** shown here[29]

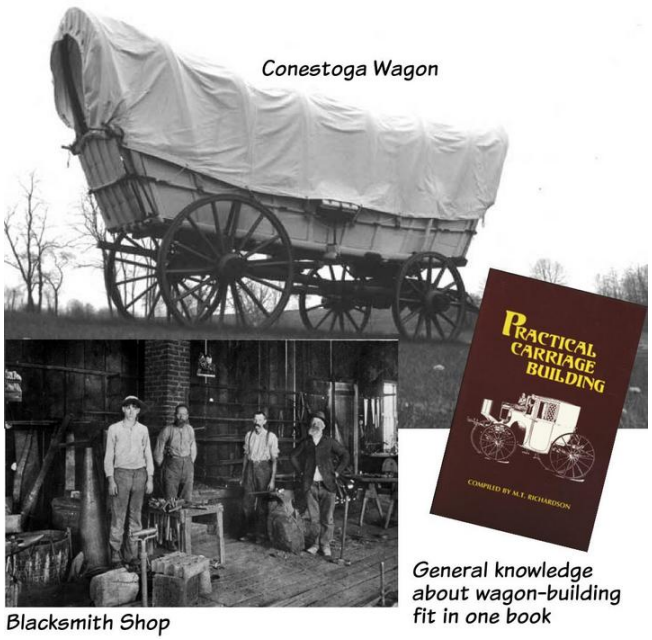
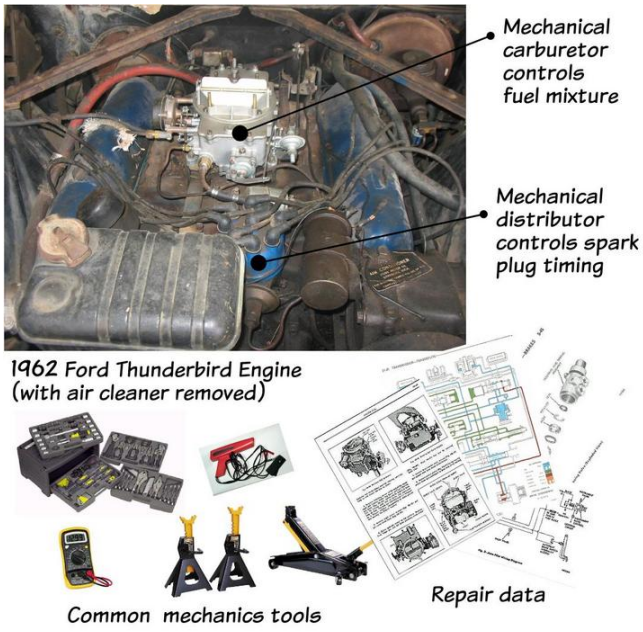



Figure 1b: Cars up until the 1970s were mechanically-tuned, meaning that they could be easily reverse-engineered if need be; repair manuals were printed by manufacturers and third parties; and a fairly inexpensive set of tools was all that was needed to repair them. This encouraged "shade tree mechanics" who maintenanced their own vehicles

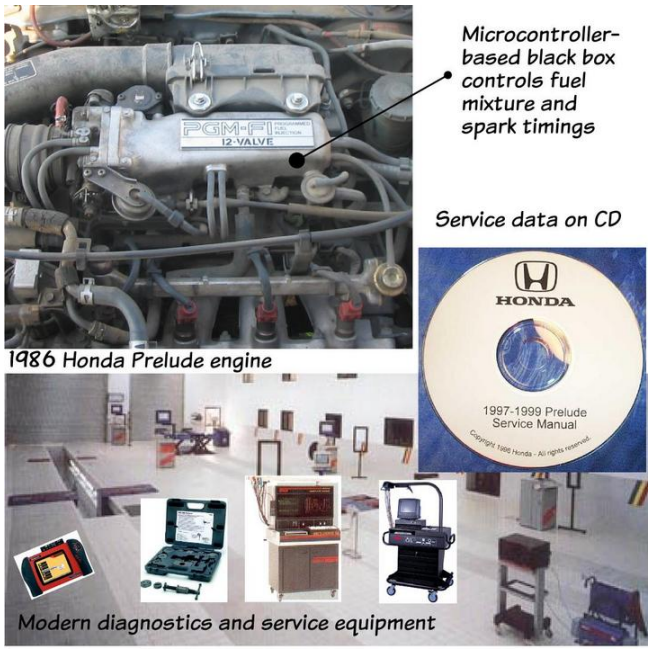



the original internal combustion engine design had of course slipped into the public domain by 1950, and high standards for originality were still in place in the patent office, so the market remained very free. It was then, and is still, fairly easy to buy third-party replica parts for these engines. Failing that, most of them could be made in a small machine shop of the type that can be found in towns all over America. The increased complexity of the automobile engine was ameliorated by the rise in standardized fasteners and tooling standards optimized for ease of repair with the small toolkits that were (and still are) widely available[2].

Then came the 1970s, with rising fuel prices, concerns about vehicle emissions (and the resulting regulations), and the availability of sophisticated digital electronics. The need for increased efficiency drove manufacturers to use integrated microcontrollers and other "black box" devices in their engine designs, primarily to improve combustion efficiency. At the same time, the American culture was seeing a boom in consumer culture, and new urban values of comfort and convenience began to supplant the old frontier ideals of self-sufficiency and independence. Car owners were going to professional mechanics instead of trying to fix cars themselves, so they no longer cared how difficult the cars were to work on. Manufacturers replaced simple standardized components in their designs with ones that were more convenient to put on in the factory, trading future repairability for lowered factory expenses. Planned obsolescence was introduced, with fixed engineering lifetimes on cars. Hoping to cash in on the public's desire for novelty, car designs changed rapidly with many proprietary customized parts for the shell of the car. For professional auto shops, these changes were relatively benign since the costs of new and more expensive diagnostic and repair tools could be distributed over many customers.

For the shade tree mechanic, however, these cars are a nightmare. They have many critical parts which cannot be reverse-engineered to work. Instead, manufacturers print "repair manuals" which are high-priced, protecting professionals at the expense of the amateurs who can no longer afford the cost of the intellectual property required to fully understand and repair their own personal vehicles. Many spe-

Figure 1c: By the 1980s, digital electronic controllers and other "black box" devices had made cars very difficult to reverse engineer, and complete service data became complex, expensive, and difficult for individuals to acquire. Cars were built to rely on complex and expensive maintenance equipment that was only practical for professional mechanics




cialized tools are needed, instead of the customary standard set of wrenches and screwdrivers. Globalized manufacturing results in cars with both metric and US units, requiring the mechanic to carry both types of tools. Disposable connectors, not widely available to end-users are used so that assemblies are not always reversibly connected. Worst of all, the "black box" microcontrollers cannot be serviced without specialized interface computers and terminals, generally only available to authorized repair shops[3]. In the end, the tooling required to fully maintenance a modern automobile *costs more than the car itself!*

Throughout this process, the end-user has been disenfranchised by the centralization of this kind of technocratic power. The essential result is exactly that of *closing the source code to the automobile*. This trend has continued right up to the present day, and the shade tree mechanic tradition—and the self-sufficiency ideal it represents—is dying because of it. This is extremely bad for those who want to see a new era of pioneering and self-sufficiency on the space frontier. Clearly, a reversal of this trend is needed.

## Repair cultures and manufacturing services

There are still cars with healthy repair cultures. The Volkswagen Beetle is perhaps the most pervasive of these. Based on a simple design, preserved almost unchanged for the many years the car was in production, the Beetle is an easy target for salvage and repair. It's easy to find parts that will match without worrying about so many "designer changes" between year-models (it uses "standard formats"), and there are many replica parts made by third party auto parts manufacturers[4].

Without spurious patents, the parts can be manufactured with few legal concerns ("free licensed" design is used), and since the Beetle's engine dates from the era of mechanically-tuned engines, it is relatively straightforward to reverse engineer by simple inspection (it uses implicitly "open source" hardware).

The Beetle remains cost-effective largely because of market-forces arising from this mix of advantages which it shares with free-licensed open-source technology. It suffers, of course, from being environmentally unsound, which is why the Beetle has not been manufactured in the United States for decades. The design has not been modernized,

Figure 2: The lowly Volkswagen "Beetle" has an impressive repair culture. Many of these vehicles are *still* maintained in the United States, despite the fact that they have not been made there since the late 1960s. Although the Beetle is not "free-licensed" in any literal sense, it has many of the same properties by historical accident, and this largely accounts for its success [Wikipedia Commons]

and the so-called "New Beetle" is based on a completely unrelated chassis and engine[5].

Naturally, of course, the Beetle would not be the most profitable vehicle to *manufacture*, since users can keep them running for so long by self-repairing them. So, is the strong repair culture model of production and maintenance "anti-capitalist"? Hardly! It merely promotes a different sector of businesses, including repair shops, how-to magazines and manuals, and after-market parts dealers. Manufacturers, machine shops, and customizers have all made the Beetle repair culture into a big commercial success for many businesses all over the world.

Furthermore, there are many societal advantages to this model. Unlike the disposable culture, the repair culture does not fill up junkyards with useless carcasses reclaimable only by recycling to component metals, and smaller service-oriented businesses are favored over multi-national conglomerates. This makes for a healthier capitalist marketplace, less pollution, greater distribution of wealth, and greater service to the end-user. This is the sort of capitalist marketplace that Adam Smith had in mind, not the world of corporate oligopolies and planned obsolescence that closed source design and overuse of patents has led us to[6].

Promoting small, localized business is also a defense for small business owners and their employees alike from the market globalization that has led to the bleed-out of manufacturing businesses from wealthier nations to the developing world: by reinventing themselves around a service model and operating closer to their target markets, they can remain competitive by focusing on the strengths of local businesses—improved service and a closer understanding of the customer.

The advantages of the Beetle were arrived at by accident more than design: there was no formal decision to free-license the Beetle, it simply arrived at this state through inexpensive design not relying on patented technologies and the passage of time, allowing patents to expire. This leaves modern automotive users with the option of a freely-repairable, but outmoded and polluting vehicle, or a modern vehicle which they are effectively disallowed from repairing.

Free-licensed open-source software provides an example of the means by which this kind of commercial success can be achieved by design: we could build hardware using open-sourced designs from the outset. For companies hoping to profit from the repair culture, this could be a great boon, but it leaves us with the problem of organizing and funding the development process, since such companies are small and distributed, whereas design and manufacturing of hardware technologies as complex as the automobile have hitherto required the use of resources only large centralized organizations have been able to marshal. Unfortunately, those organizations, who generally rely on profit from sales to recoup the costs of development, are unlikely to embrace the free-design model. Must small business manufacturers alter their plans to suit this corporate agenda, or is there an alternative form of organization that can accomplish the same goals in a way that better suits both customer and vendor?

## Deconstructing the corporation

When considered as a social order rather than as an individual institution, a large corporation approximates a centrally planned economy with all of its flaws. Despite the great expense of traditional management, decisions have historically tracked only loosely with the needs of customers, resulting in enormous gaps between consumer expectations and corporate performance—exactly the properties that killed most centrally planned state economies in the 20th century. Only the absence of a viable alternative structure would seem to explain the continued success of corporations.

From the same perspective though, a community-based enterprise, consisting of self-organizing free agents is a free market, with all of the resulting advantages of accurately and naturally tracking participants' needs and wants and requiring far less overhead to maintain. In a bazaar development community, customers are directly involved in the innovation process, and thus customer needs are freely translated into designs. All manufacturing is then "outsourced" directly to the same small manufacturers who would previously have contracted from the corporations. In this way, the free bazaar replaces the proprietary cathedral of the corporate design center.

When all of the manufacturing capacity has been outsourced to small vendors and the design process is under the direct control of the customers, what value is left in the corporate management structure? Management decisions are then only an expensive, redundant, and error-prone noise-source which interferes with the signal of the bazaar's free market

Figure 3a: A corporate enterprise is strongly top-down controlled, with a marketing subunit to collect information from a consumer market which is assumed to be external. Selection of the corporate management is restricted to a small class of people (possibly overlapping with the market) who have bought shares in the company. Design centers and production centers are controlled rigidly by management, which wholly owns them. Some production subunits (and occasionally design and marketing subunits as well) are contracted, but total control is ensured through exclusive contracts, non-disclosure agreements and other legal instruments designed to keep power in the hands of the corporation's management. Classic management problems with large corporations are the same as with state-controlled economies—primarily due to faulty information transfer and non-optimal group decision-making

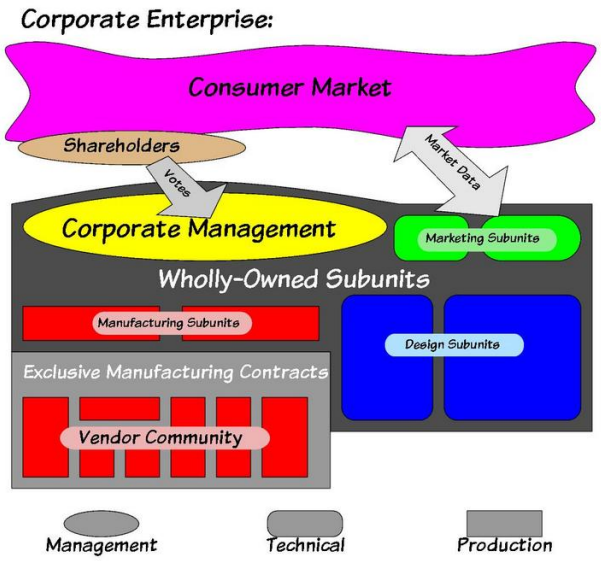
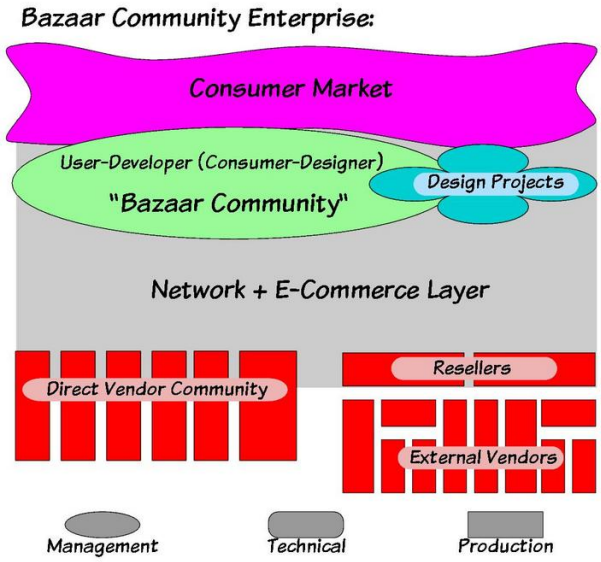



Figure 3b: A free-design bazaar community is a free market of ideas controlled by the fraction of the consumer market that is willing to spend the time and thought on it. As a self-motivated "interestocracy" it can efficiently replace both the design centers and the management decision-making system. Relying on free-licenses allows the manufacturing subunits to operate in a free market without binding contracts and the resulting inefficiencies. The trust and commerce relationships between the interacting parties are essentially peer-to-peer and can be moderated by an internet application




of ideas and sucks resources out of the system. The sole remaining true utility of the corporate structure—as a medium of exchange of information and trust between subunits—is a routine, deterministic process which can be replaced much more efficiently, accurately, and cost-effectively by a thin, automated network layer.

With the baggage of the actual manufacturing process removed, the products of the design center become information products, freeing design centers to use free-licensed methodology, leaving only the problem of implementing effective software tools for managing the distributed design process and maintaining the bazaar community[7]. Obstacles to contracting the manufacturing service directly from the end customer are merely technical: how to mediate the exchange of resources required and resolve trust faults in the process[8].

## Surviving ephemeralization

The factory has become increasingly like the printing press, and the printing press has become increasingly like the computer printer: manufacturing is trending slowly towards the ideal of a Star Trek "replicator" which can make anything given sufficient energy and a blueprint. Even the modest progress achieved in this direction in the form of "3D Printers" and other highly-flexible rapid-prototyping machines is likely to have a significant impact on engineering and manufacturing industries.[9]

While that may seem like a perfect situation for the consumer, it threatens change for the manufacturing industry. For the foreseeable future, the existing trends in manufacturing, towards reduction of both the marginal and capital costs of production will mean that, increasingly, the capital cost of a product is *not* in the capital equipment required to manufacture it, but in the effort required to design it. For the design centers, this suggests the viability of applying free software methods.

For small manufacturers, this could be an opportunity to contract directly for the production of freely-designed hardware to the end customers and the developers who designed it, cutting out the intermediary of the large corporate contractor, and putting the smaller business into a position of much greater freedom. Without non-disclosure or exclusivity agreements, patent licenses, and tightly binding contracts to worry about, the small manufacturer has much less buy-in and therefore less risk with producing free-licensed hardware. Likewise, the greater availability, greater flexibility, and lower cost of capital manufacturing equipment will mean that the manufacturer can become sufficiently agile to meet the short-run and prototype manufacturing needs of a bazaar-style community, in much the same way as the publishing industry has been able to adapt with print-on-demand and short-run publications for the rapidly moving target of the free software community.

> The advantages of the Beetle were arrived at by accident more than design

Large vendors organized around 20th-century-style mass production of consumer goods should increasingly see competition from these smaller, more agile vendors. Gradually, the disadvantages of centralized control will begin to outweigh the advantages of centralized production as the latter become weaker and weaker due to the manufacturing process approaching the replicator ideal.

## Vendor niches

At present, the direct-sale market for small vendors is usually a narrow niche of hobbyists with relatively little money to spend and fairly modest requirements. But as a free-design community is encouraged to grow, increasing developer ambitions will create more and more opportunities for supply and service vendors who can support this development process and produce free-hardware products for end customers.

### First generation vendors

Even in the beginning, hardware design bazaars will expand commercial opportunities for many vendors, including:

- Existing service manufacturing and repair businesses, such as machine shops, welding fabricators, and printed-circuit board printers who are already able to find customers, but could greatly expand their business by doing prototype work for free projects[10].
- Retail parts providers who currently specialize in the hobbyist market. The bazaar would provide opportunities for both direct sales and sales through resellers, allowing much more room for parts from smaller manufacturers[11].
- Tool and die manufacturers and retailers[12].

Mitigating the risk-averse society

In the space community, as in many other fields of endeavor, many people bemoan the current "risk-averse" society in which it is almost illegal to take certain risks. More accurately, the legal code has moved to blaming manufacturers for injuries or other harm caused by use (or misuse) of their products. Such liability risks are among the most frequently cited reasons for small businesses not to make cutting-edge technology available.
Free-licensed design, however, offers two potential defenses against this kind of threat:
The first is "full disclosure". Since the basis for manufacturer responsibility is arguably that the manufacturer is in a privileged information position, the complete disclosure of a product in the form of open-source design may be considered a fair way to transfer the risk onto an end-user willing to assume it.
The second defense is the "experimental" designation. It's an indication that the full legal risk for the safety of the product must be assumed by the purchaser. It is routine for OEM companies rather than end-users to assume risk in this way, but if end-users purchase in the same way, the same rules should apply to them.
In the most serious case we can further remove the liability issue by collecting releases, using a nominal membership fee and a training process to make the "public" into "members" of an organization. This is how the Pacific Rocket Society (and presumably other similar groups) reduce their legal liability risk—only "members" are allowed at launches, and members sign releases assuming the risk[25].

- Testing services which provide the primary means of establishing quality control in the manufacturing process (this is how manufactured product specifications can be verified).
- Professional consulting services, such as engineering design review. Although bazaar development primarily relies on user testing and user review of designs, there will remain significant demand for credentialed review of plans, especially as projects move from experimental to fully-integrated designs.

The existence of free-licensed design bazaars creates a new market for such companies whereby they can sell much more directly to the end customer, and remove their dependency on few large contracts; or from another point of view, the bazaar marketplace *is* a large customer, perhaps analogous to a government agency, but with far less red tape.

### Later generation vendors

Once a free-design community becomes more established, there will arise niches for businesses which both supply and are supplied-by the community, manufacturing free-designs arising from the community, both for use in more complex designs and for direct sale to end-users. These end-users, just like the self-sufficient frontiersmen are really consumer-manufacturers, just like the user-developers that make up most of the free software user community. In the same way, they assume responsibility for the use of the design, which they have selected, paying only for the service of manufacturing—a very different bargain from that of existing consumer retailers. It is also possible, of course, for vendors to assume greater responsibility and market products through conventional distribution channels, incurring the resulting liability costs. This gives two additional vendor niches:

- Service manufacturing for the end customer (in the US at least, this sometimes requires the product to be a "kit" to be assembled by the end customer—one can buy "experimental" aircraft in this way)[13].
- Conventional sales of free-licensed consumer products. This involves increased liability and so will probably be more expensive, but can reach a larger market.

An eventual goal of free-hardware development, should be to create a full vertical range of free, commoditized and standardized components, starting with things like LSI CPU chips[14] and concluding with entirely free-designed systems (like desktop computers, automobiles, or spacesuits). At the point that this is achieved, the products of the free-design community can begin to enter the mainstream of consumer products. Long before that, they will expand the alternative market for adventurous consumers who are willing to risk using experimental products.

## Vendor needs

This new marketplace will expand the opportunities for independent services and service-manufacturing vendors, but doing so requires recognition of the problems such companies will face in staying in business. The biggest problem is marketing. For a small auto machine shop "marketing" may currently mean only a sign, a well-chosen location near auto repair shops or a race track, and word-of-mouth advertising. For a one-off PC board fabricator, it may be little more than a website linked to hobbyist web pages (electronic "word-of-mouth" advertising)[10].

The internet, of course, immediately suggests automated marketing opportunities, and just as E-Bay created a viable online marketplace for one-of-a-kind merchandise, putting the boutique business model online[15], there have been attempts to build online service brokering sites to advertise services, including manufacturing[16]. However, such sites are generally organized exclusively around service brokering, and so they are very supplier-oriented. This makes using them something of a chore to use—the sort of thing only a purchasing agent will do. Which leaves only the same customers that were available before the internet. This is no better than simply automating the Thomas Register, which Thomas has already done.[17]

There are, however, a number of practical problems with selling to customers through an online system, which vendors will naturally fear, and which therefore will need to be addressed in a system designed for contracting the manufacture of free-licensed components:

### Non-payment

- Fears: Payment won't be made even after the non-recoverable expense of manufacturing has been invested.

Free patents or no patents?

As other writers have already argued effectively, the only thing patents are doing for innovation in the 21st century is drowning it in lawyers[26]. The reduced manufacturing friction of an ephemeralized hardware economy makes this drag force painfully visible. But what to do about it?

There are several proposed defenses against patents as a threat to innovation: One is to publish "prior art" for free-designs, so that they can't be co-opted by monopolizers. Another is to define a "free-licensed patent" defending against patent restrictions just as the GPL does against copyright restrictions.

There is considerable disagreement, however, about whether such a patent license is possible[27]. Some previous attempts at this have been decidedly non-free, and unpopular with the free-licensed software community. It's also possible to simply ignore patents—they are difficult and expensive to enforce, so it is likely that most never will be. Furthermore, in the US, ignorance of a patent is considered reason for clemency in cases of patent violation.

It is of course, possible to lobby for the elimination of patents, but this is unlikely to work until inventors can demonstrate the folly of patents by proving a viable alternative exists. Meanwhile, one of the techniques of living with patents will have to be used. In all probability, future innovation will resolve this issue, but until then, means such as the W3C Patents Policy[28] should be provided to allow both.

- Wants: Assurance that payment will be made.

This is what escrow payment is for, and why we need to have specifications that a third party can confirm.

### Loss of control

- Fears: Agreements with brokering service will interfere with business's independence.
- Wants: To pick and choose their own contracts.

This is why we must have an automated system for browsing specifications and finding suitable contracts, and why restrictions on basic vendor memberships must be kept minimal.

### Too much overhead

- Fears: Any automated brokering system will take too much time to use, detracting from primary business.
- Wants: Either very easy, or someone else does it.

The automated system for browsing specifications should be easy enough for non-skilled users to take advantage of it (like a search engine). Finally, however, there is an opportunity for companies which resell the services of companies that don't sell directly to the bazaar, such resellers will eventually come into competition with their suppliers as those suppliers learn of the site, providing for a gradual evolution from mostly reselling to mostly direct sales as the bazaar grows.

### No customers

- Fears: Site will be useless, no one will buy.
- Wants: Low risk of participation, site oriented to attract customers.

Existing service brokering sites are primarily for that purpose only, attracting only the most aggressive customers. The bazaar on the other hand, is primarily a design site for project developers and donors (who are the customers) to interact. Therefore the site will naturally bring vendors into contact with customers.

### No scale economies

- Fears: Relying on short-run business will not be profitable enough.
- Wants: Ideally to batch products into larger runs for economy's sake.

The promotion of standard designed components encourages projects to reuse already created components, for which tools and dies have already been made. The company which made the tooling investment will be in position to charge the least for future manufacturing.

### Liability

- Fears: They will be sued by end-user customers for non-performance or even hazardous behavior of products.
- Wants: Indemnification of design responsibility: since vendors don't design end-products they can't be responsible for performance.

This is intrinsic to the idea of service manufacturing—the final integrator is the one responsible for performance. In the bazaar model, this is unusual because the final integrator is the customer. All sales must be on an experimental basis, and vendors should be able to limit sales to "responsible individuals" through such means as waivers and proof of age.

### Patents

- Fears: They will be sued for patent-violation.
- Wants: Knowledge of patent infringement possibilities.

There is no way to absolutely remove this threat, as it is a problem endemic to the patent system, but the insistence on free-licensed designs and the free publication of those designs as prior art provide some assurance of design freedom.

## A free marketplace

The initial marketplace of the free-design community is contiguous with the existing hobbyist and do-it-yourselfer market. It is such technically-literate customers from whom free designs will be generated, and to whom most sales will be made, especially at the outset. The example of the shade tree mechanic stands as a cautionary tale of the dangers of proprietary and closed design, but it is important to realize that do-it-yourselfers are not gone. Despite their problems, the shade tree mechanics still persist, albeit mostly with older cars.

There are amateur machinists[18] and electronics hobbyists[19], model plane enthusiasts[20] and sport rocketry amateurs[21], aircraft homebuilders[22] and robotics competitors[23]. Altogether, amateur technologists are a powerful group. Even the reaches of outer space are not beyond them,

Figure 4: Amateur do-it-yourselfers are a much more powerful and capable group than you might think. Even spaceflight is an active area of amateur development, as the successes of radio amateur satellites have shown


as proven by over fifty amateur satellites launched, starting in 1961, all primarily created and maintained by radio amateurs. One of the most recent, AO-40, has even gone into high Earth orbit to return images of the Earth from space[24].

Complete self-contained, end-user-operated replication machines won't arrive for a long time. However, we don't need to wait for a literal "replicator" to see economic effects from such technology trends. Meanwhile, the falling costs of replication can be made less visible and more distributed. The acquisition of capital and coverage of marginal costs can be pushed further out—closer to the end-user or consumer, in the form of direct contracting to service manufacturing businesses. The cost of entry to design and manufacturing processes alike can be reduced, creating a system which empowers the end-user rather than centralized manufacturing centers.

## Bibliography

[1] Covered wagons (http://www.endoftheoregontrail.org/wagons.html), construction (http://www.nps.gov/jeff/overland_wagon.html), and blacksmithing (http://www.saskschools.ca/~gregory/blsmith.html)

[2] From Sears (http://www.sears.com/), for example.

[3] But for serious amateurs, Software for Cars (http://www.obd-onboarddiagnostics.com/)

[4] Bug Haus (http://www.bughaus.com)

[5] History of VW "Beetle" (http://en.wikipedia.org/wiki/Volkswagen_Beetle)

[6] Adam Smith, An Inquiry into the Nature and Causes of the Wealth of Nations (http://www.gutenberg.org/etext/3300), 1776.

[7] Narya Project (http://narya.net)

[8] Narya Bazaar (http://bazaar.narya.net)

[9] 3D Printers: Solid freeform fabrication (http://www.mae.cornell.edu/ccsl/research/sff/) and Cubital's Soldier 5600 (http://www.wired.com/wired/archive/2.05/eword.html?pg=7)

[10] For example, Pad2Pad (http://www.pad2pad.com/)

[11] For example, Mouser Electronics (http://www.mouser.com/)

[12] For example, Sherline Machine Tools (http://sherline.com/)

[13] For example, Kit-built planes (http://www.homebuilt.org/kits/kits-vend.html)

[14] Open Cores (http://www.opencores.org/) and LART (http://www.lart.tudelft.nl/)

[15] e-Bay (http://ebay.com)

[16] For example, Elance (http://www.elance.com/)

[17] ThomasNet (http://www.thomasnet.com/)

[18] Amateur Machinists (http://sherline.com/resource.htm)

[19] EDA Forum (http://www.edaboard.com/)

[20] Model Airplane News (http://www.modelairplanenews.com/)

[21] Dallas Area Rocket Society (http://www.dars.org/)

[22] Homebuilt Homepage (http://www.homebuilt.org/)

[23] Battlebots (http://www.battlebots.com/) and Botball (http://www.botball.org/)

[24] Amsat (http://www.amsat.org/)

[25] Pacific Rocket Society (http://v-serv.com/prs/)

[26] Faré Rideau, Patents Are An Economic Absurdity (http://fare.tunes.org/articles/patents.html), and references therein.

[27] Links in All the Way Open (http://open.narya.net)

[28] W3C Patent Policy (http://www.w3.org/Consortium/Patent-Policy-20040205/)

[29] M. T. Richardson, Practical Carriage Building (http://www.astragalpress.com/practical_carriage_building.htm), 1892.

## Copyright information



## About the author

Terry Hancock is co-owner and technical officer of Anansi Spaceworks (http://www.anansispaceworks.com), dedicated to the application of free software methods to the development of space.

# Free software and Latin America

## The challenge is often political rather than technical

David Sugar

In 2002, the then–U.S. Ambassador to Peru John Hilton (http://www.wired.com/news/business/0,1367,54141,00.html) delivered a threatening letter (http://www.linuxjournal.com/article/6244) to the Peruvian congress on behalf of a powerful American private interest. The letter, which was leaked to the press, stated that the Microsoft Corporation and its chairman Bill Gates disapproved of Peruvian politicians debating a proposed law, Special Bill 1609 (http://linuxtoday.com/news_story.php3?ltsn=2002-05-06-012-26-OS-SM-LL), which favored the use of free software in public administration. Hilton warned its passage would harm U.S.-Peru relations. The bill was quietly dropped after then–Peruvian president Alejandro Toledo was invited by Microsoft chairman Bill Gates to personally receive a donation for a Toledo controlled Peruvian foundation.

In 2002, the then–U.S. Ambassador to Peru John Hilton delivered a threatening letter to the Peruvian congress, using his public office to act on behalf of a very powerful American private interest

This incident was an early salvo in a brewing international conflict over free software. As nations of the world increasingly turn to free software to cut costs and promote local development, some powerful North American commercial interests have responded by outright bullying. Sometimes they have done so by proxy, using public servants like Ambassador Hilton. Other times they have threatened legal action to intimidate (linktext) outspoken critics such as Brasil's National Institute of Information and Technology (ITI) President Sergio Amadeu da Silveira, who compared Microsoft business practices to that of drug dealers. Sometimes, they have threatened governments directly, like last November (2004), when Microsoft CEO Steve Ballmer threatened (http://www.theregister.co.uk/2004/11/18/ballmer_linux_lawsuits/) to sue Asian governments who choose to use free software.

### What is this challenge about?

Part of what distinguishes free software commercially from proprietary software is a matter of licensing. While all software is under copyright restriction, commercial proprietary software is often licensed under terms that create additional restrictions, such as limiting where one can use such software and who may be allowed to use it. Often, proprietary commercial software include licenses which explicitly deny users: the right or ability to modify software to fit their needs or access their own data, the right to speak about the functionality of the software they have purchased, or the right to resell it to others when they no longer wish to use it. In contrast, free software expressly asserts and grants these fundamental rights through licensing, and does so in a way

that enables others to fully reclaim these rights such as by providing source code.

Free and proprietary commercial software have co-existed uneasily for a long time in many parts of the world. However, I believe the reason that certain private North American commercial interests have responded directly in Latin America is that many of the nations there have chosen to promote the use of free software specifically in public administration. There is already a long history for the support and use of such software in Brasil by the Workers party, starting from the days when they controlled the state government of Rio Grande do Sul and instituted private/public sector partnerships through projects such as procergs (http://olinux.uol.com.br/artigos/264/print_preview.html). Most recently the government of President Luiz Lula Da Silva has chosen to use free software solutions built around GNU/Linux exclusively, in a project to make computers available to the poor (http://query.nytimes.com/gst/abstract.html?res=F40614FD395B0C7A8EDDAA0894DD404482), as recommended by MIT (http://www.computerworld.com/softwaretopics/os/story/0,10801,100494,00.html) this past March.

Free software in public administration is not just about software for special government programs such as digital inclusion for the poor. This is a battle about the purchase and use of all software by national governments and the terms such software will be provided under. This about the procurement of servers and database applications used to house government data. This is also about the software that will be purchased and used on the desktops of government office workers every day, and whether they will continue to purchase and use Microsoft Windows and Microsoft Office under the terms of a monopoly supplier, or free software alternatives such as GNU/Linux and OpenOffice.

As Latin American governments increasingly use free software, suppliers will need to adapt to provide it. Private industries which interact with government will also be effected to remain compatible, and provide additional private markets for those vendors. All of these create a national economic environment that certain companies, such as Microsoft, would need to change in order to fully participate in.

I believe the reason that certain private North American commercial interests have responded directly in Latin America is that many of the nations there have chosen to promote the use of free software specifically in public administration

## Why do Latin American nations choose free software?

One reason that Latin American governments are promoting free software is the question of initial cost. In Brasil, they expect to save over one billion dollars (http://www.brazzil.com/2004/html/articles/apr04/p136apr04.htm) annually through the use of free software and the elimination of license fees. Many other Latin American governments are of course keenly aware of the cost benefits of free software. In some countries, such as in Peru and Argentina, they have tried passing special procurement laws to more rapidly increase the adoption of free software in government. In Venezuela, the use of free software in public administration is now supported directly by President Hugo Chavez (http://venezuelanalysis.com/news.php?newsno=1457).

While it is true that the total cost of using software is not represented in the purchase price or license fees alone, most other factors also tend to favor free software and better explain the potential for large cost savings through its use. One reason is commercial free software will often work on existing and older hardware rather than requiring new hardware to be purchased. Another is that since proprietary commercial software publishers depend on the number of licenses they can sell, it is often desirable to require as many additional software sales to perform a given level of work as possible. It therefore comes as no surprise to me that I often find the same workload that can be done, for example, with a typical GNU/Linux system may require three or four times as many proprietary servers, which also represents additional hardware and support costs.

The use of free software can also result in lower costs through the absence of monopolies. One cannot achieve

a monopoly in free software in part because there can always be another free software publisher that can supply the same goods at a lower cost should this occur. This is in fact one of the main reasons that governments prefer using free software instead of proprietary commercial software: when money is spent on proprietary software, only a small proportion of that money goes towards funding useful services and software development, as a large part of it goes as a monopoly rent to the shareholders of the proprietary software company. On the other hand, in the world of free software, where there are no such monopolies, money spent on free software is good for creating jobs and hence offers other direct and local economic benefits. In Latin America, money that is spent on proprietary commercial software serves mainly to make already-rich foreign software publishers even richer.

> While it is true that the total cost of using software is not represented in the purchase price or license fees alone, most other factors also tend to favor free software and better explain the potential for large cost savings through its use

In trying to create a market for or to promote the use of free software, many Latin American countries, such as Peru, have often chosen to do so through procurement laws, which cover how a government will purchase goods and services. These laws typically state the terms of purchase that a government will use. Often they are designed to prevent bribery, and to make the process of government purchase transparent. This is often done through the use of competitive bidding. Competitive bidding allows products created by different manufacturers and publishers to compete on providing the same service, and by doing so, prevents the government from being forced to rely on a sole source supplier. Propriety commercial software, by its very definition and through the rights it takes away from users, is software which can only come from a single supplier.

In providing opportunities for Latin American citizens to directly participate in the development and worldwide commercial software market locally, free software offers incentives for forming a local software industry that can then compete on an equal basis with that of any other advanced country in the world. What we often forget is that software does not require expensive plants or high capital investment to develop. Software may only require people who are free to use their skills and natural talents. Certainly, the nations of Latin America can and do produce people with such talents and skills. Free software means these people can practice these skills for their own benefit and the benefit of their society as a whole without having to look for work in or migrate to foreign lands. By choosing to procure free software, the national government can directly encourage this.

## So what's the problem?

If Latin American countries choose to create an economic environment that accepts participation by free software, existing corporations need not be excluded. Companies like Microsoft could choose, for example, to change the way they license their existing products. They are also free to adapt and offer services based on existing free software already in the marketplace. Instead of competing in these new markets, some companies have responded by trying to make it impossible for Latin American governments to choose and use free software at all.

These companies not only resort to bullying, but also lobby the U.S. government to modify free trade treaties and use international organizations to include conditions that try to make it impossible for Latin American nations to choose alternative products or develop local markets.

I have, in particular, seen the World Intellectual Property Organization (WIPO) used in this way to promote the private commercial interests of wealthy corporations. WIPO is often used to promote treaties and laws which handle ideas, culture, and literature as if they were physical property. WIPO has also been used to export the North American corporate notions of pharmaceutical and software patenting to developing nations. Private corporations then use these same treaties to enforce existing North American patent monopolies, thereby preventing the development of competitive local industry. Another example of market control through trade treaties is the "IP rights chapter" of the Free Trade Area of The Americas (FTAA) treaty.

One of the ways I have seen Latin American countries respond to WIPO and other patent bearing treaties has been to band together with other developing nations around the world to help promote a development agenda (http://

www.eff.org/IP/WIPO/dev_agenda/) for WIPO and bring it into harmony with the wishes of the WIPO general assembly. Yet powerful American and European commercial interests have chosen to use the WIPO chair to explicitly bar NGOs that represent the interests of developing nations from attending or participating in WIPO discussions on a development agenda, even those organizations already duly certified and recognized with observer status.

## Conclusion

The people of Latin America, of all people, surely must understand well what corporate bullies are. Last century many nearby Caribbean nations were routinely invaded by marines as part of the banana wars (http://experts.about.com/q/673/3343542.htm) to prop up the interest of specific North American corporations such as United Fruit.

While last century's bullies came with tanks and guns, the bullies of this new century come now with the laws and treaties they wish Latin Americans to adopt. These laws and treaties undermine the heritage and most basic rights that Latin American citizens enjoy; and not for the benefit of Latin America, but instead, once again for the benefit of private North American corporate interests. The right to innovate is not a privilege to be restricted to a tiny minority, it's not even a right specific or exclusive to the question of free software alone, but is a basic and fundamental right every human being must be free to enjoy.

## Copyright information



## About the author

David is an active maintainer for a number of packages that are part of the GNU project (http://www.gnu.org). He has also served as the voluntary chairman of the FSF's DotGNU steering committee (http://www.dotgnu.org).

# Are your systems *OPEN* for business?

## *Our customers' are!*

For over 10 years, OCI has been helping organizations build their mission critical infrastructure with open architectures using standards-based OO technologies.

OCI provides the following commercial-grade products to the open-source community:


- ACE - The high performance C++ portability layer combining socket-level performance with type-safe programming, across a wide range of operating systems.
- TAO - An open-source C++ CORBA 2.6 implementation with the widest range of services and span of platforms of any ORB.
- JacORB - The open-source Java CORBA 2.4 implementation.
- MPC - An open-source cross-platform build tool - "script once, build many"
- OVATION - A tool for instrumenting & visualizing CORBA systems behavior.

**Can't wait to go open?** OCI offers a comprehensive set of migration support services that will help your organization minimize risk, cost, and time when moving to these open-source solutions.

To learn more about these products & services, visit: www.theaceorb.com

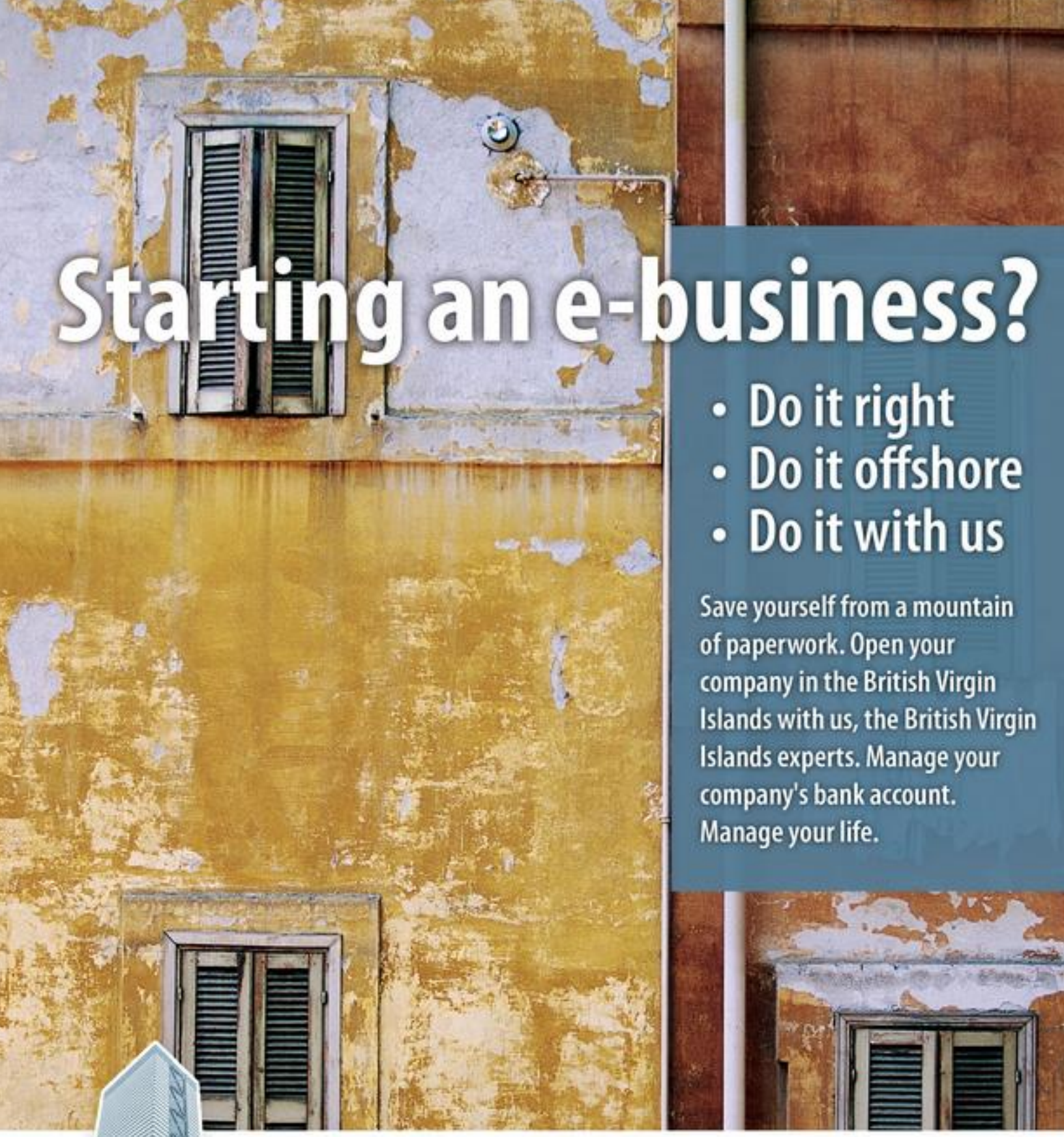
**Product Development • Consulting • Educational Services**

Object Computing, Inc. (OCI)

St. Louis, MO Headquarters: +1 (314) 579-0066
Phoenix, AZ Office: +1 (480) 752-0042

**www.ociweb.com**

OBJECT COMPUTING, INC.



# Starting an e-business?

- Do it right
- Do it offshore
- Do it with us

Save yourself from a mountain of paperwork. Open your company in the British Virgin Islands with us, the British Virgin Islands experts. Manage your company's bank account. Manage your life.

Di Bernardo Massimo
OFFSHORE

90 Main St - Road Town, Tortola
British Virgin Islands

http://www.dbm-offshore.com
info@dbm-offshore.com