RV Backup and Media Server

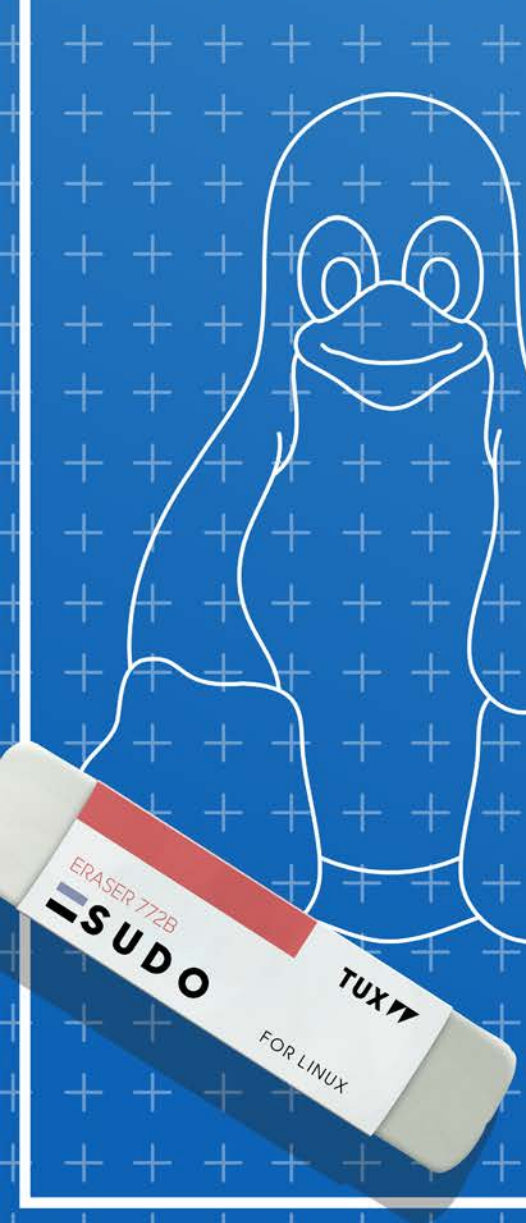
Build a Custom Minimal Distro

Open Hardware and IoT

LINUX JOURNAL

Since 1994: The original magazine of the Linux community

DO-IT-YOURSELF



Issue 287 June 2018

www.linuxjournal.com

CONTENTS

JUNE 2018
ISSUE 287

PHOTO CREDIT: JOY RANKIN

CONTENTS

LINUX JOURNAL (ISSN 1075-3583) is published monthly by Linux Journal, LLC. Subscription-related correspondence may be sent to 9597 Jones Rd, #331, Houston, TX 77065 USA. Subscription rate is $34.50/year. Subscriptions start with the next issue.

ARTICLES

AT YOUR SERVICE

SUBSCRIPTIONS: *Linux Journal* is available as a digital magazine, in PDF, EPUB and MOBI formats. Renewing your subscription, changing your email address for issue delivery, paying your invoice, viewing your account details or other subscription inquiries can be done instantly online: http://www.linuxjournal.com/subs. Email us at subs@linuxjournal.com or reach us via postal mail at *Linux Journal*, 9597 Jones Rd #331, Houston, TX 77065 USA. Please remember to include your complete name and address when contacting us.

ACCESSING THE DIGITAL ARCHIVE: Your monthly download notifications will have links to the different formats and to the digital archive. To access the digital archive at any time, log in at http://www.linuxjournal.com/digital.

LETTERS TO THE EDITOR: We welcome your letters and encourage you to submit them at http://www.linuxjournal.com/contact or mail them to *Linux Journal*, 9597 Jones Rd #331, Houston, TX 77065 USA. Letters may be edited for space and clarity.

SPONSORSHIP: We take digital privacy and digital responsibility seriously. We've wiped off all old advertising from *Linux Journal* and are starting with a clean slate. Ads we feature will no longer be of the spying kind you find on most sites, generally called "adtech". The one form of advertising we have brought back is sponsorship. That's where advertisers support *Linux Journal* because they like what we do and want to reach our readers in general. At their best, ads in a publication and on a site like *Linux Journal* provide useful information as well as financial support. There is symbiosis there. For further information, email: sponsorship@linuxjournal.com or call +1-281-944-5188.

WRITING FOR US: We always are looking for contributed articles, tutorials and real-world stories for the magazine. An author's guide, a list of topics and due dates can be found online: http://www.linuxjournal.com/author.

NEWSLETTERS: Receive late-breaking news, technical tips and tricks, an inside look at upcoming issues and links to in-depth stories featured on http://www.linuxjournal.com. Subscribe for free today: http://www.linuxjournal.com/enewsletters.

LINUX JOURNAL

EDITOR IN CHIEF: Doc Searls, doc@linuxjournal.com

EXECUTIVE EDITOR: Jill Franklin, jill@linuxjournal.com

TECH EDITOR: Kyle Rankin, lj@greenfly.net

ASSOCIATE EDITOR: Shawn Powers, shawn@linuxjournal.com

CONTRIBUTING EDITOR: Petros Koutoupis, petros@linux.com

CONTRIBUTING EDITOR: Zack Brown, zacharyb@gmail.com

SENIOR COLUMNIST: Reuven Lerner, reuven@lerner.co.il

SENIOR COLUMNIST: Dave Taylor, taylor@linuxjournal.com

PUBLISHER: Carlie Fairchild, publisher@linuxjournal.com

ASSOCIATE PUBLISHER: Mark Irgang, mark@linuxjournal.com

DIRECTOR OF DIGITAL EXPERIENCE:
Katherine Druckman, webmistress@linuxjournal.com

GRAPHIC DESIGNER: Garrick Antikajian, garrick@linuxjournal.com

COVER DESIGN: Carty Sewill

ACCOUNTANT: Candy Beauchamp, acct@linuxjournal.com

Linux Journal is published by, and is a registered trade name of,
Linux Journal, LLC. 4643 S. Ulster St. Ste 1120 Denver, CO 80237

SUBSCRIPTIONS
E-MAIL: subs@inuxjournal.com
URL: www.linuxjournal.com/subscribe
Mail: 9597 Jones Rd, #331, Houston, TX 77065

SPONSORSHIPS
E-MAIL: sponsorship@linuxjournal.com
Contact: Publisher Carlie Fairchild
Phone: +1-281-944-5188

Our Immodest Ambitions

Doc Searls is a veteran journalist, author and part-time academic who spent more than two decades elsewhere on the *Linux Journal* masthead before becoming Editor in Chief when the magazine was reborn in January 2018. His two books are *The Cluetrain Manifesto*, which he co-wrote for Basic Books in 2000 and updated in 2010, and *The Intention Economy: When Customers Take Charge*, which he wrote for Harvard Business Review Press in 2012. On the academic front, Doc runs ProjectVRM, hosted at Harvard's Berkman Klein Center for Internet and Society, where he served as a fellow from 2006–2010. He was also a visiting scholar at NYU's graduate school of journalism from 2012–2014, and he has been a fellow at UC Santa Barbara's Center for Information Technology and Society since 2006, studying the internet as a form of infrastructure.

Some guidance along our road to greatness.

By Doc Searls

In a February 2018 post titled "Worth Saving", I said I'd like *Linux Journal* to be for technology what *The New Yorker* is for New York and *National Geographic* is for geography. In saying this, I meant it should be two things: 1) a magazine readers value enough not to throw away and 2) about much more than what the name says, while staying true to the name as well.

The only push-back I got was from a guy whose comment called both those model pubs "fanatically progressive liberal whatever" and said he hoped we're not "*planning* to emulate those tainted styles". I told him we weren't. And, in case that's not clear, I'm saying it here again. (For what it's worth, I think *The New Yorker* has some of the best writing anywhere, and I've hardly seen a *National Geographic* outside a doctor's office in decades.)

Another commenter asked, "Is there another publication that you'd offer up as an example to emulate?" I replied, "Three

come quickly to mind: *Scientific American*, the late *Dr. Dobb's* and *Byte*. Just think of all three when they were at their best. I want *Linux Journal* to honor those and be better as well."

Scientific American is the only one of those three that's still alive. Alas, it's not what it once was: the most authoritative yet popular science magazine in the world—or at least, that's how it looked when my parents gave me a subscription when I was 12. Back then I wanted to read everything I could about science—when I wasn't beeping code to other ham radio operators from my bedroom or otherwise avoiding homework assignments.

Today, *Scientific American* is probably as close as it can get to that legacy ideal while surviving in the mainstream of magazine publishing—meaning it persists in print and digital form while also maintaining a constant stream of topical stories on its website.

That last thing is the main work of most magazines these days—or so it seems. As a result, there isn't much difference between *Scientific American*, *Smithsonian*, *Wired*, *Ars Technica* and *Inverse*. To demonstrate what I mean, here are stories from those five publications' websites. See if you can guess (without clicking on the links) where each one ran—and which one is a fake headline:

- "Rare Mammoth Tracks Reveal an Intimate Portrait of Herd Life"
- "The New Distributed Ledger That Out-Blockchains Blockchain"
- "If You Want to Know How to Stop School Shootings, Ask the Secret Service"
- "We Were So Very Wrong About the Size of the Andromeda Galaxy"
- "America's Secret Ice Base Won't Stay Frozen Forever"

The problem with all of them isn't only that they could run anywhere, but that they are all just content—or seem to be, no matter how good they might actually be.

(While they can be very good, one example of how they often aren't is that I had to fix the spelling in one of the five real headlines.)

Here's what I said in **"The Problem with Content"** (in the March 2017 issue of *Linux Journal*):

> Back in the early '00s, John Perry Barlow said "I didn't start hearing about 'content' until the container business felt threatened." *Linux Journal* was one of those containers—so was every other magazine, newspaper and broadcast station. Today, those containers are bobbing around in an ocean of "content" on the Internet. Worse, the stuff inside the containers, which we used to call "editorial", is now a breed of "content" too.
>
> In the old days, editorial lived on one side of a "Chinese wall" between itself and the publishing side of a newspaper or magazine. The same went for the programming and advertising sides of a commercial broadcast station or network. The wall was transparent, meaning it was possible for a writer, a photographer, a newscaster or a performing artist to see what funded the operation, but the ethical thing was to ignore what happened on the other side of that wall. Which was easy to do, because everything on the other side of that wall was somebody else's job.

Today that wall has been destroyed by the imperatives of "content production", which is the new job of journalists and everybody else devoted to "generating content" in maximum volumes, all the better to attract "programmatic" advertising.

By doing only the old-fashioned kind of advertising that simply sponsors our work and carries no tracking, we're free to zero-base something different and better than you'll see elsewhere. And that applies both here inside the magazine and in other media as well, starting with the website.

So far I see three requirements for becoming as good as we want to be.

First is featuring great writers you know by name and who you expect to produce

great stuff. [I'm not sure *OMNI* magazine (1978–1995) was great, but it sure tried hard. I was part of that, contributing humorous pieces to early issues. When that failed to become a full-time gig, I did what many other journalists do when they reach their rope's end: I started an ad agency. That grew into one of the biggest in Silicon Valley before I got pulled back to journalism again in the 1990s—by *Linux Journal*.] We already have some of those in our stable of veteran contributors, but we want more.

Second is by being as fun, practical and authoritative as we can on Linux topics. A nice model for that, among successful contemporary publications, is *Make*. We should be for Linux what *Make* is for the maker movement.

Third is by going deep. I want us to drill down on topics that are important and relevant to our readers, but not covered elsewhere—or not covered well enough.

As always, we invite guidance from readers toward all of those. ■

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

LETTERS

Re: "How Wizards and Muggles Break Free from the Matrix"

Yesterday I was wondering why there isn't a more organized resistance to the CopperTopping of humanity, as I tried to watch Amazon Prime on my "Smart" TV without telling Google more about me. I remember—circa 2000—when I was the only person I knew who was concerned. Now there is a fairly large contingent actively concerned. But, effectively, nobody is doing anything—because effectively nobody knows how to.

Oh, yeah, there is complaining. And there are occasional expressions of concern from a various and small contingent of companies and legislators. And then there is your article; although notably your article starts with "How" but—again—doesn't say anything specific enough to deserve its title. So, in that regard, your article is a bit of the circle-jerk itself, even if its expression is backed by good intentions.

The techie Priesthood still holds nearly all the specialized knowledge required to even attempt to break away without going "third world" (forgoing a mobile phone, forgoing a TV, forgoing using a credit card). WHY doesn't *Linux Journal* begin giving specific HOW advice and, better yet, use its pages to cultivate groups of people required to liberate through TEACHING how to do? A step-by-step for those concerned but not sufficiently in the priesthood.

A great article, to start with, would be specific recommendations on "Smart" TVs that best protect privacy and specific, direct means to short-circuit Google, under its myriad names and minions, from all our devices. I don't ask for perfection here; only a reasonable and sustained start with some path forward for the hundreds of millions of humans who are pretty smart, pretty concerned, yet not hardware-geeks enough to know how to break out of the twisty-turny maze that keeps changing through time and between models? I'd love to join THAT organization—I'd even subscribe to THAT magazine.

—Gnat

Doc Searls replies: Thanks, Gnat.

I'll address your uppercase HOW and TEACHING concerns separately.

Above a bulleted list in that piece, I say "Here's a punch list of what we need to do." I should have added "—and HOW" to that sentence. Because, if you go to the links in that list, you'll find lots of HOWs. Specifically:

- The link behind "Customer Commons' work on terms individuals can proffer as first parties, and others—primarily sites and services—can agree to, in ways both can record in case a dispute arises" goes to that work. The GDPR in Europe all but requires these kinds of agreements, by the way. I'll be writing about that again soon too.

- *Linux Journal* will be the first site in the world to agree to a first party Customer Commons term proffered by its readers.

- Personal control of how we are known (if at all) to others in the online world is coming to be called self-sovereign identity. That link goes to a white paper on the topic by the Sovrin Foundation.

- Since one way sovereign (independent, liberated, free, autonomous) individuals can present their choice of identifiers (or absence of those, since identifiers are usually not required for any two parties to interact or do business) is through *verifiable claims*, and systems for recording those tend to include blockchains, I have that link going to a search for verifiable+claims+blockchain.

- For "Pay what we want, where we want it, and how we pay for whatever we want, in our own ways", I point to an approach called EmanciPay. I invite geeks to build it.

- I explain "Change what all the companies we deal with know about us—such as our phone number, our last name or our email address, *in one move*", in a

piece titled "Customers need scale". Same goes for "Call for service or support in one simple and straightforward way of our own."

- "Express loyalty in our own ways" goes to a piece titled "Loyalty means nothing if customers don't have their own ways of expressing it." This is do-able too.

- The link for "a true Internet of Things" goes to Phil Windley's post titled "The Internet of My Things Needs the API of Me". Phil and his coding colleagues (many of whom are students of Phil's at BYU) have already created some of the code required. I'd like to see more geeks involved.

- Behind the "giving each (thing) its own cloud" is a post of mine titled "The answer is #CFT: Clouds For Things", which was inspired by work already done by Phil and friends.

- "Own and control all our health and fitness records" goes to a project called HIE of One, which is explained here. It's led by Adrian Gropper, an MD (Harvard) and tech wizard (MIT).

- The link behind "generously sharing helpful facts about how we use their products and services" goes to a post titled "Market intelligence that flows both ways". In it I unpack the huge implications of code that's already in the world.

- The links behind "have wallets and shopping carts that are our own" go to old posts that wizards should revisit before working on either of those tools. We've always needed them, but wizards keep designing ones that work only in corporate silos. Time to fix that.

To your point about TEACHING: sure, we should do that. We also should remember that lots of others are doing that already. This search gets more than a half million results.

On the other hand, one good invention can get adoption by dozens of millions in a short time.

Twenty years ago, Marc Andreessen told me "All the significant trends start with technologists." (That was in an interview I did with him for *Linux Journal*, titled "Betting on Darwin".) Lots of significant trends followed work he and his colleagues did as technologists then.

One of our jobs here is to encourage development of inventions that cause necessity for everybody—wizards and muggles alike. Not everything we do is "a reasonable and sustained start with some path forward for the hundreds of millions of humans". But that is pretty much what we've tried to do during the last 24 years. We've succeeded in some cases and failed in others.

I believe "Breaking Free of the Matrix" is the most important work we've encouraged since our early efforts to spread the Linux word. That's why I wrote that piece. Again, I invite readers to help us succeed with that.

I'm Pleased

I read Doc Searls' "Privacy Is Still Personal" editorial in the May 2018 issue and agree completely. The Wintendo generation (kids, now parents who grew up with Nintendo and Windows) doesn't get this. I want privacy. I'm only a little person, but others could do great damage to me if they got hold of my numbers (SS#, drivers license, DOB, etc.). Do you remember *PC Mag* when it was almost an inch thick? That's how you attract. Stop targeting.

I will fully support you folks. Let me know how I can help.

—Roy Niemann

Doc Searls replies: Thanks for your support, Roy. We appreciate it.

About privacy, everybody wants it, or they wouldn't wear clothes. The fact that

we don't have it yet online is a matter of absent tech, not absent consciousness. Invention is the mother of necessity. When we develop good personal privacy tech, people will need it. Nobody needed a smart phone before Apple and Google made the market for them. Now almost nobody can get along without one. Here at *Linux Journal*, we're doing what we can, in our own ways, to foster that market—and we need all the help we can get!

Second, by coincidence, I wrote for the first inch-thick *PC Magazine*, back when it started in 1982. I still have a copy around here somewhere. My contribution was a story on Ken Uston's Professional Blackjack, a game for IBM PCs and Apple IIs that taught card counting. (Geek item: it was written, as I recall, in Forth.) I also edited the game manual, which I'm amazed to find still selling on Amazon. (I also just found that you can run it in emulation, through a link at Archive.org.)

Nextcloud

Regarding Marco Fioretti's "Nextcloud 13: How to Get Started and Why You Should" in the May 2018 issue, I have been using Nextcloud since I switched to it from OwnCloud a little more than a year ago. It is an excellent app for me! I am running 13.02 on Ubuntu 17.10 with MySQL and PHP 7.1. One thing I would like to point out, and that you may want to cover in an article relatively soon, is backing up the installation with recovery in mind. I was running my instance on an old Toshiba Satellite (about six years old), and all was well until it was not. Fortunately, I use storeBackup, which is aimed at an external USB drive (0.5 Terabyte). A nightly database dump (MySQL) along with the files in the Nextcloud data directory almost got me back to where I needed to be. As it turned out, the passman addon would not function properly based on my recovery strategy, and it was a bit tough to get everything synced back up with regard to devices and files. (I bought a newer used Toshiba laptop as my new server and went from Bodhi 4.0 to Ubuntu 17.10.) I did get everything back up to snuff, but had I done a better job of using storeBackup, it would have taken a lot less time. I don't think that would have made a difference with passman, but it certainly would have with Calendar, Contacts and Files! One other note:

there is an addon that is very useful to me that you may want to mention for those with health issues, and that is the DICOM viewer. I always get a DVD of my medical images, and the DICOM viewer works very nicely for showing medical images to a new provider who does not have immediate access to one's medical records from other providers. At least they can see what I am talking about, and that sometimes saves me a return trip and another office visit co-pay. Very cool! Thanks for writing this up! A good article!

—Wes Wieland

Marco Fioretti replies: Thanks for your compliments! I'm obviously happy you liked the article.

What you say about the Dicom viewer is great. It's a real-world example of the power of open standards, and personal/self-hosted clouds to overcome the limits and annoyances of proprietary, non-interoperable systems. I must remember to mention your case next time I speak on those topics.

About covering backups in an article "relatively soon", *Linux Journal* and I are planning more Nextcloud articles, but the details aren't all defined yet, so we'll see. One general lesson that your email illustrates well is that Nextcloud backup depends heavily on which *combination* of applications are installed, and also on if/how they interact with *other* Nextcloud installations. So, I'll need to think about how to explain this well in one article, but I definitely will keep your suggestions in mind!

Librem Review

Wow! Regarding Shawn Powers' review of the Librem 13v2 in the May 2018 issue, where were you last week when I was shopping for a new laptop? I have been eyeing the Librem for a long time and wanted to get an honest, full review before pulling the trigger. Your review rocked, and it made me happy to know I chose well last week when I ordered one. Now if it will just ship!

I have used Linux on the server for many, many years, and I've tried to transition to the desktop many times over the years. I have been a Mac guy until last November when I built a desktop, and I've been living in Linux ever since. Year of the Linux Desktop indeed! Now if my Librem will get here, I will be complete.

I will say the only thing I am really missing is a Calendar/Contacts solution to rival Mac's iCloud integration. I don't want to give Google all my info and would like to have desktop app solutions. For now, it's Thunderbird, but I would love to hear other solutions if any exist.

Thanks for keeping *LJ* going.

Not just a fan, a subscriber!

—Russ Demarest

Great Review of Librem 13v2

I loved Shawn's review of the Purism Librem 13v2 laptop in the May 2018 issue of *Linux Journal*. And, what perfect timing! I am shopping around for a new laptop, and I already was looking at the Librem 13.

Shawn's review was a great summary of the Librem 13's features, especially the keyboard and trackpad—always a concern when buying a new laptop. I also appreciated Shawn's discussion of the quirks, which was an honest review about what to expect.

Thanks for doing these reviews.

—Jim Hall

Everything You Need to Know about the Cloud and Cloud Computing

I just read Petros Koutoupis' "Everything You Need to Know about the Cloud and

Cloud Computing, Part I". Reading it was a pleasure. Normally I look at articles like this and know that I should read them, but then I think I'd rather pull my fingernails out. Why? Because most people cannot write a technical article and make it interesting. I was so engaged that it encouraged me to look the author up online and connect with him so I wouldn't miss future articles. That is a very special skill set. I look forward to reading more.

—Paul Byrne

Surprised

In *Linux Journal* March 2018, I read the article "Looking Back: What Was Happening Ten Years Ago?" by Glyn Moody, and I was very much surprised to see that he writes this:

> Another name figured quite prominently in my columns of ten years ago: Firefox. Along with the Apache web server, it is one of open source's great success stories, taking on Microsoft's slow and bloated Internet Explorer, and winning. At the time, it seemed like Mozilla might be able to build on the success of Firefox to strengthen the wider open-source ecosystem. Mozilla is still with us, and it's doing rather well financially, but it has had less success with its share of the web browser market. As a graph on Wikipedia shows, Firefox's ascent came to a halt around 2010, and its market share has been in decline pretty much ever since. That's not to say that Mozilla is not important to the world of free software—partly because of its solid finances, it does much valuable work in many related fields. But Firefox has become something of a niche browser, used mostly by die-hard supporters.

And in the same issue, you cite Firefox as the "Best Web Browser":

Best Web Browser

- Firefox: 57%
- Chrome: 17%
- Chromium: 7%

- Vivaldi: 6%
- Opera: 4%
- Brave: 3%
- qutebrowser: 2%

How should I interpret this?

—Wolf-Dieter Klotz

Glyn Moody replies: Thanks for your email. In fact, there's no contradiction between the browser statistics I quoted and Firefox's position in the Readers' Choice Awards 2018.

The first is an estimate of Firefox's share of the overall market share, including all browsers, all operating systems and all users. In that world, Firefox has seen a steady decline to its present levels, largely driven by the rise of Chrome.

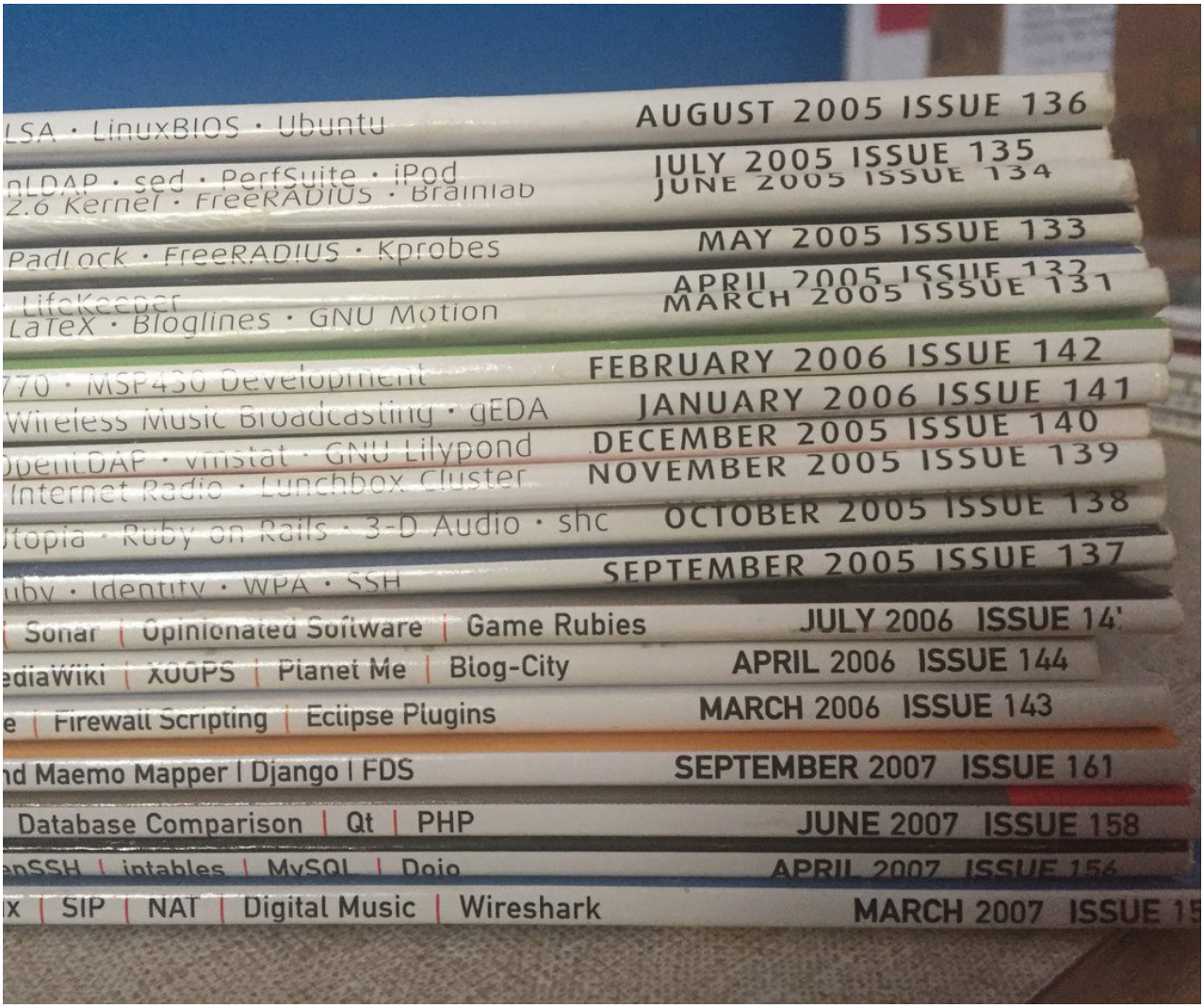
The second figure, 57%, represents the proportion of *Linux Journal* readers who think Firefox is the best browser. That concerns a much smaller sample size, which obviously has important differences from the general user population.

Linux Journal readers are likely to place a greater emphasis on open-source, privacy protection and related matters. Firefox also regards those things as priorities, which helps explain why it is much more popular among *Linux Journal* readers than among the general population, who may not value them so highly.

From Social Media

Kyle Rankin @kylerankin: In an upcoming @linuxjournal article, I inventory the various DIY #RPis and other single-board computers running around my house. I was kind of shocked when I realized the total was 10 different computers.

Luke Triantafyllidis @ltriant: Unpacking boxes and came across these old copies of @linuxjournal from when I used to buy them at university. Gonna be



fun to read through them again!

Editor's note: you can now access Linux Journal's *online HTML archive—24 years of articles in one easy-to-search resource, and it's free to subscribers.*

***In response to our thanks for subscribing to* Linux Journal:**

Adam Dymitruk @adymitruk: Replying to @BrideOfLinux @linuxjournal: Considering what people spend on coffee everyday, I didn't have to think twice. How much do you value freedom, openness and decentralization in an era

of Facebook? The big guys are trying really hard to keep platform out of the freedom and privacy debate.

In response to our news story about a new stable release of GIMP being released after six years of development:

Richard Gaskin @FourthWorldSys: The best graphics package, held back all these years by an unfortunate name. ■

SEND *LJ* A LETTER *We'd love to hear your feedback on the magazine and specific articles. Please write us here or send email to ljeditor@linuxjournal.com.*

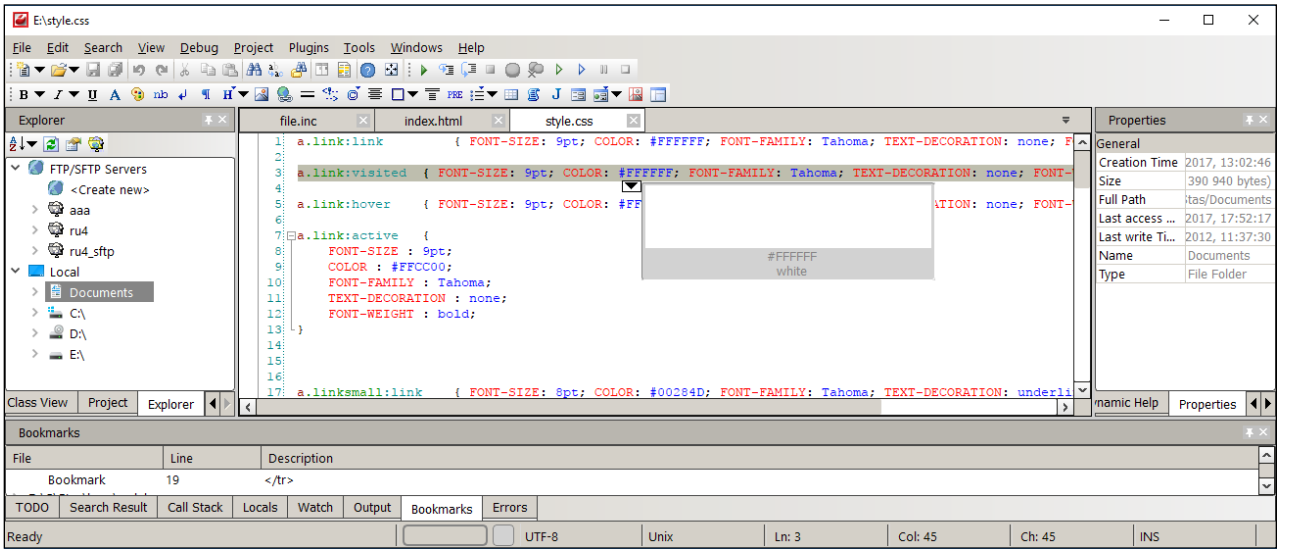
PHOTOS *Send your Linux-related photos to ljeditor@linuxjournal.com, and we'll publish the best ones here.*

FOSS Project Spotlight: the Codelobster IDE—a Free PHP, HTML, CSS and JavaScript Editor

The Codelobster free web language editor has been available for quite some time and has attracted many fans. It allows you to edit PHP, HTML, CSS and JavaScript files, and it highlights the syntax and provides hints for tags, functions and their parameters. This editor deals with files that contain mixed content easily as well.

If you insert PHP code in your HTML template, the editor correctly highlights both HTML tags and PHP functions. The same applies to CSS and JavaScript code, which is contained in HTML files. The program also includes an auto-completion function, which greatly speeds up work for programmers and eliminates the possibility of errors.

The Codelobster IDE provides contextual help on all supported programming languages, and it uses the most up-to-date documentation, downloading it from official sites so you quickly can get a description of any HTML tag, CSS attribute, PHP or JavaScript function by pressing the F1 key.



The built-in PHP debugger allows you to execute PHP scripts step by step, sequentially moving through the lines of code. You can assign check points, view the process of the work of loops, and monitor the values of all variables during the execution of a script.

You can view the HTML templates directly in the editor, highlight the interesting elements on a page and explore the associated CSS styles. The HTML and CSS inspector works by following the well known FireBug principle.

Other useful functions and features of the IDE include:

- Pair-highlighting of parentheses and tags—you'll never need to count parentheses or quotation marks; the editor takes care of it for you.

- Highlighting of blocks, selection and collapsing of code snippets, bookmarks to facilitate navigation on edited files, recognition and building of the complete structure of PHP projects—all of these functions ensure easy work with projects of any scale.

- Support for 17 user-interface languages, including English, German, Russian,

Spanish, French and more.

- The program works on the following operating systems: Windows 7, Windows 8, Windows 10, macOS, Linux, Ubuntu, Fedora and Debian.

The professional version of the Codelobster IDE provides programmers with even more features.

For example, you can work with projects on a remote server with the use of the built-in FTP client. You can edit the selected files, preview the results and then synchronize the changes with the files on the hosting side.

In addition, the professional version includes an extensive set of plugins:

- Fully implemented support for JavaScript libraries, such as jQuery, Node.js, AngularJS, BackboneJS and MeteorJS.

- A large set of extensions that helps when working with PHP frameworks, such as CakePHP, CodeIgniter, Laravel, Phalcon, Smarty, Symfony, Twig and Yii.

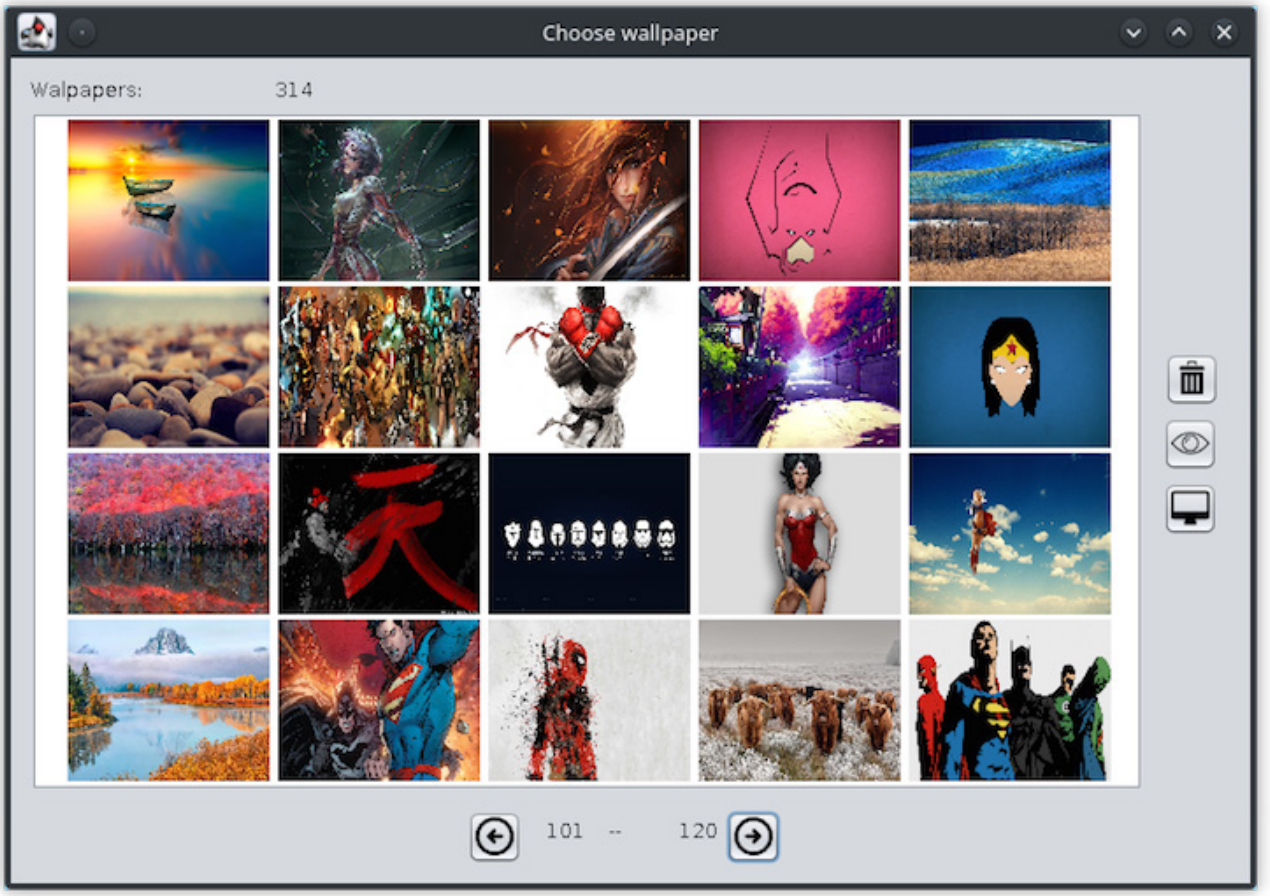
- Plugins for working with the most popular CMSes, including Drupal, Joomla, Magento and WordPress.

You can download and install any framework directly from the program without being distracted from your main tasks.

Generally speaking, I should note that during a year of work, our team had no complaints about the editor. The Codelobster IDE works fast, doesn't hang and even allows us to work with large PHP projects.

You can download the Codelobster IDE from the official website.

—Stanislav Ustimenko



FOSS Project Spotlight: WallpaperDownloader

Are you bored with the look of your desktop? Are the wallpapers that come with your distro enough for you? WallpaperDownloader is a graphical application that will help you customize your desktop and find wallpapers automatically.

WallpaperDownloader allows you to download, manage and change your favorite wallpapers from the internet. It is open source (GPL3) and totally free. Simply type in some keywords, enable the providers to include (up to six), select the download policy, and WallpaperDownloader does the rest.

WallpaperDownloader's main features include:

- Users can select keywords for matching desired wallpapers across different sources.
- Currently, six providers are implemented for searching.
- Different download policies are implemented.
- Preferred resolution for the search can be defined.
- The maximum size for downloaded directories can be changed.
- Wallpapers can be classified as favorites or not favorites.

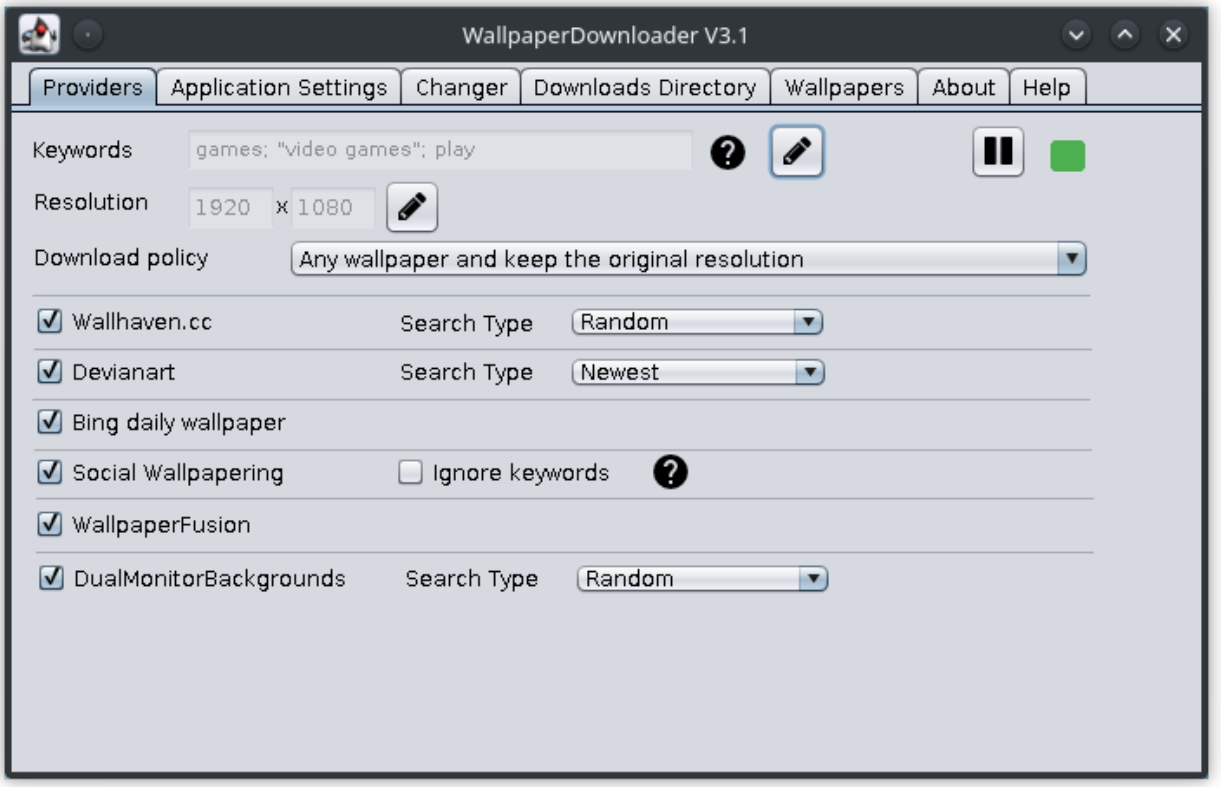


Figure 1. Selecting Providers

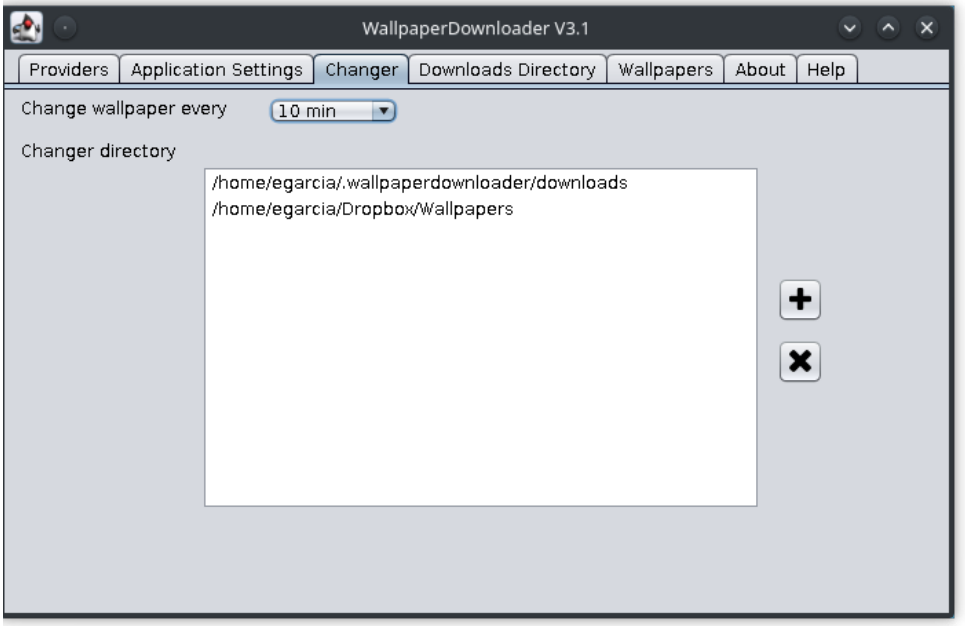


Figure 2. WallpaperDownloader's Changer

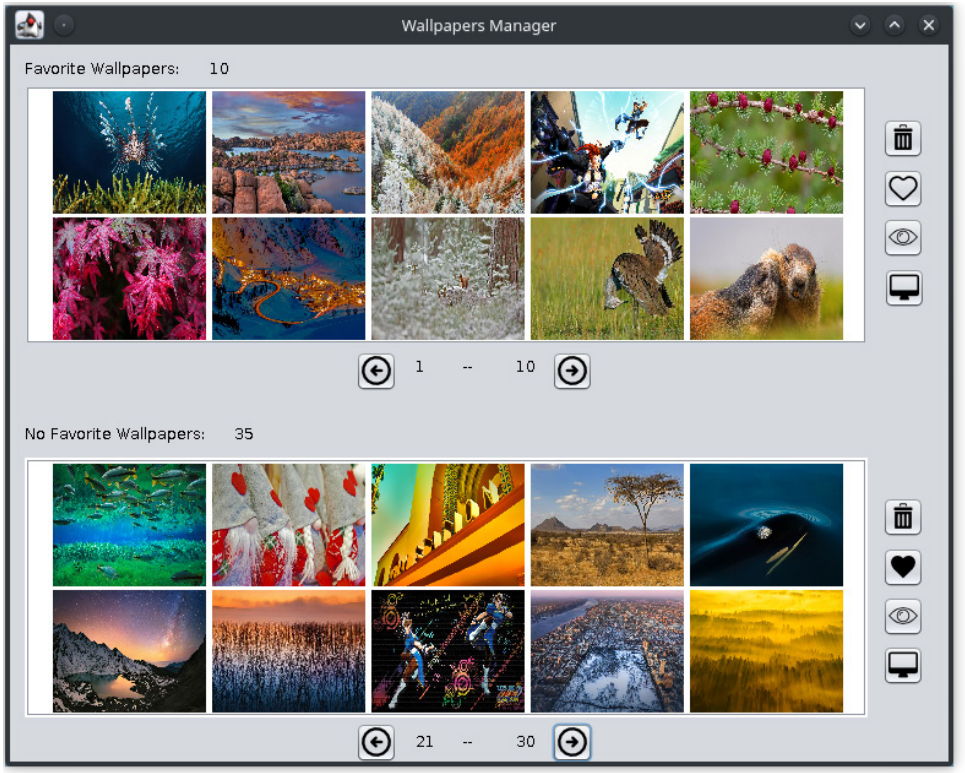


Figure 3. Wallpaper Manager

- Favorite wallpapers can be moved to another location with a single click. This is very nice if you have a directory for storing images (for example, a directory in Dropbox).

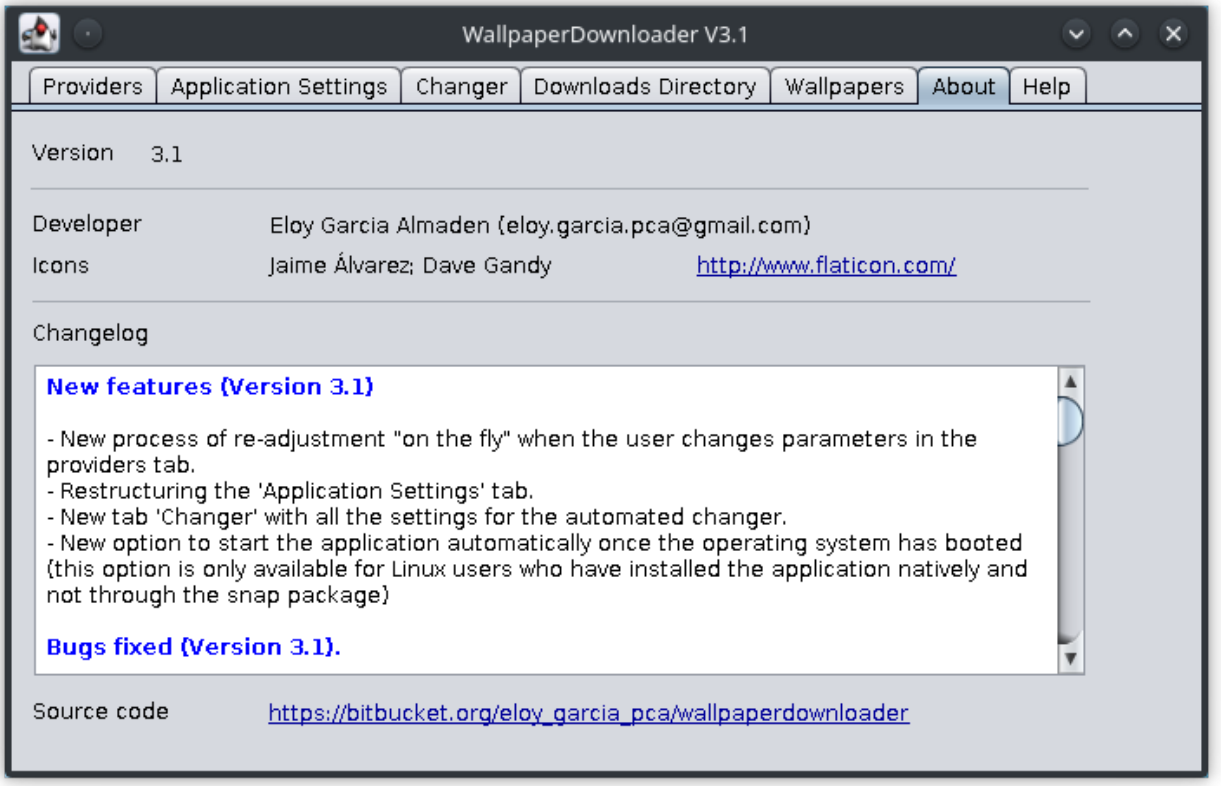


Figure 4. WallpaperDownloader Info and Changelog

- WallpaperDownloader is translated into English and Spanish so far.
- It implements an automated "changer" for changing the wallpaper randomly every X minutes. You can define as many directories as you want.
- A system tray icon is implemented (for desktop environments that support this feature) with quick actions.

WallpaperDownloader supports several Desktop Environments: MATE, GNOME Shell, Cinnamon, Budgie, Pantheon, Unity, KDE Plasma 5.8 or greater and XFCE.

Installation

You can install WallpaperDownloader using different methods depending on your distribution.

Arch Linux It is in the AUR repository. Just install it from there:

```
yaourt -S wallpaperdownloader
```

Ubuntu, Derivatives and Linux Distros with snapd WallpaperDownloader is available from the Ubuntu Software Center via snap package (just search for it). If you want to install it via terminal, type:

```
sudo snap install wallpaperdownloader
```

Caveats: the snap package fully supports GNOME Shell, Unity, Budgie, Cinnamon, Pantheon and MATE desktop environments. If you are using KDE Plasma 5 (version 5.8 or greater) or XFCE and your distro of choice is Ubuntu, installation via official PPA is recommended.

Ubuntu and Derivatives via PPA There is an official PPA repository for installing WallpaperDownloader in Ubuntu (16.04 and greater) and derivatives natively. This is the preferred method for enabling all the features of the application, and it's recommended for KDE Plasma 5 and XFCE users. Open a terminal and type:

```
sudo add-apt-repository ppa:eloy-garcia-pca/wallpaperdownloader
sudo apt update
sudo apt install wallpaperdownloader
```

I hope you enjoy the application. Feedback, contributions and help with translating the GUI are always welcome!

—Eloy Garcia Almaden

Patreon and *Linux Journal*

PATREON

Together with the help of *Linux Journal* supporters and subscribers, we can offer trusted reporting for the world of Open Source today, tomorrow and in the future. To our subscribers, old and new, we sincerely thank you for your continued support. In addition to magazine subscriptions, we are now receiving support from readers via Patreon on our website.

LJ community members who pledge $20 per month or more will be featured each month in the magazine. A very special thank you this month goes to:

- Zachary Klein
- David Breakey
- NDCHost.com
- Mostly_Linux

BECOME A PATRON

Drawing Feynman Diagrams for Fun and Profit with JaxoDraw

I've been covering chemistry software in my last few articles, so this time, I decided to move to physics and introduce a package called JaxoDraw. In physics, there's a powerful technique for visualizing particle interactions at the quantum level. This technique uses something called Feynman diagrams, invented by physicist Richard Feynman. These diagrams help visualize what happens when one or more particles have some kind of interaction. I say one or more because a single particle could spontaneously kick out other particle/anti-particle pairs and then swallow them back up again. Needless to say, quantum physics is weird.

When first developed, theoretical physics was mostly done either with pen and paper or on a chalkboard. Not much thought was given as to how you could render these drawings within a document being written on a computer. JaxoDraw is meant to help fill in that gap in document layout and provide the ability to render these drawings correctly and give output you can use in your own documents.

JaxoDraw is written in Java, so it should run under almost any operating system. Unfortunately, it isn't likely to be in the package repository for most distributions, so you'll need to download it from the project's website. But, because it's packaged as a jar file, it's relatively easy to run.

Download the binary package, unpack it on your machine, and then you'll want to open a terminal and change directory to the location where you unpacked

JaxoDraw. You can start it simply by typing the following:

```
java -jar jaxodraw-2.1.0.jar
```

This opens a blank workspace where you can start your diagram. On the left-hand side of the window, you'll see a palette of all of the available drawing elements that you can use to generate your diagram.

To see what's involved, let's draw an electron interacting with a photon. This happens when energy is absorbed or emitted by an electron. Since you're looking at an interaction, you'll want to start by selecting the vertex button from the palette and then draw one in the window. Coming into this vertex will be a fermion line for the electron and a photon line for the incoming electromagnetic energy. The interaction happens at the vertex, with a second fermion line coming out the other end. You can continue adding more elements, including loops or bezier lines, and you also have the

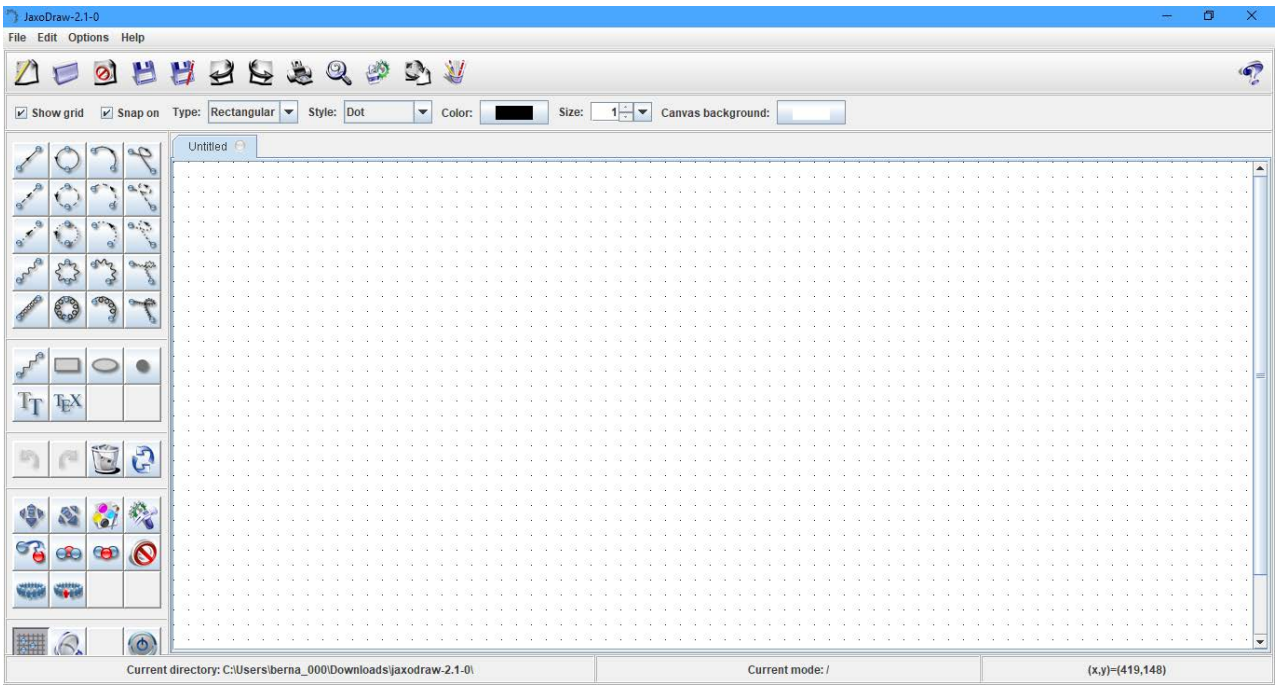


Figure 1. When you first open JaxoDraw, you see a blank workspace where you can start diagramming your quantum particle interaction.

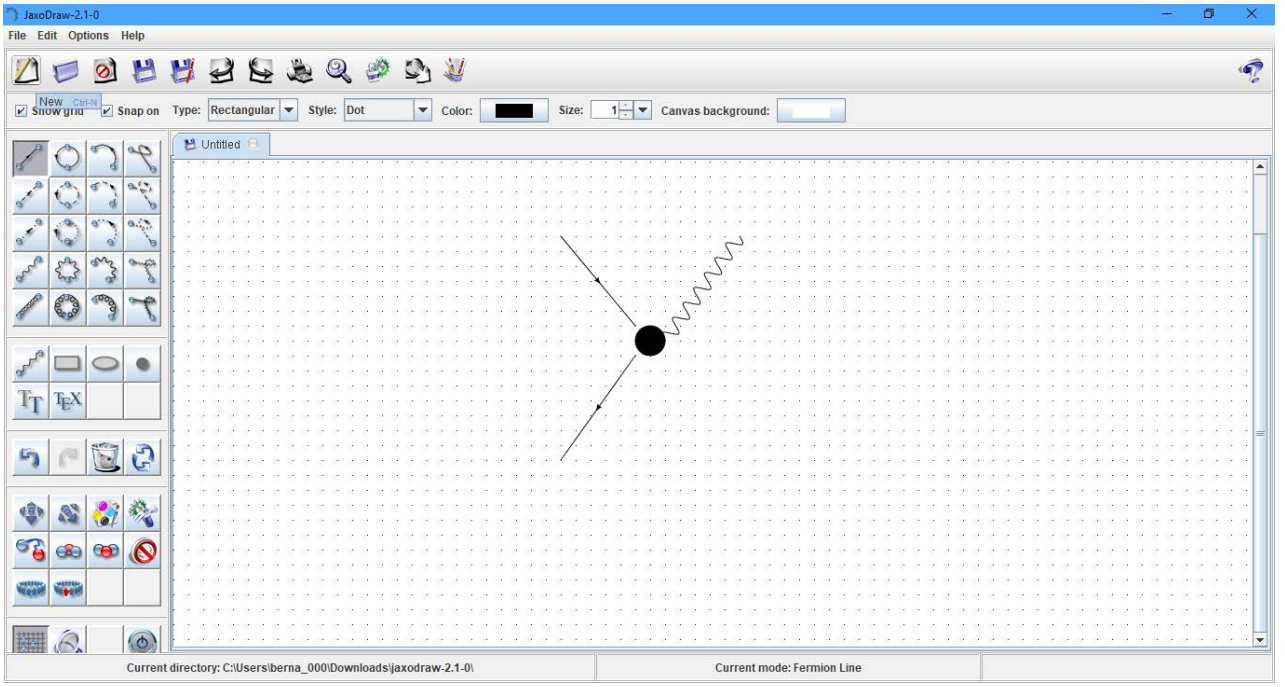


Figure 2. A basic interaction is an electron absorbing a photon's worth of energy and continuing on.

choice of other particle types, such as scalar particles, ghost particles or gluons.

If you click the edit button in the function palette on the left-hand side and then select one of the elements of your diagram, you'll get a pop-up window where you can edit parameters including the position, size and location of any extra parts, like arrow heads or the color used for a given element. This edit window varies based on what parameters can be changed for a given element.

JaxoDraw also includes a plugin architecture, where you can extend its functionality. A list of current plugins is available at the plugin website, which currently consists of other exporters to generate other file types of your diagrams. There is also a set of instructions on how you can create your own plugins.

Once you download the plugin file, click the menu item Options→Plugin Manager to pop up a dialog window where you can install the plugin within your installation of JaxoDraw.

Figure 3. You can edit several parameters of the elements within your diagram.

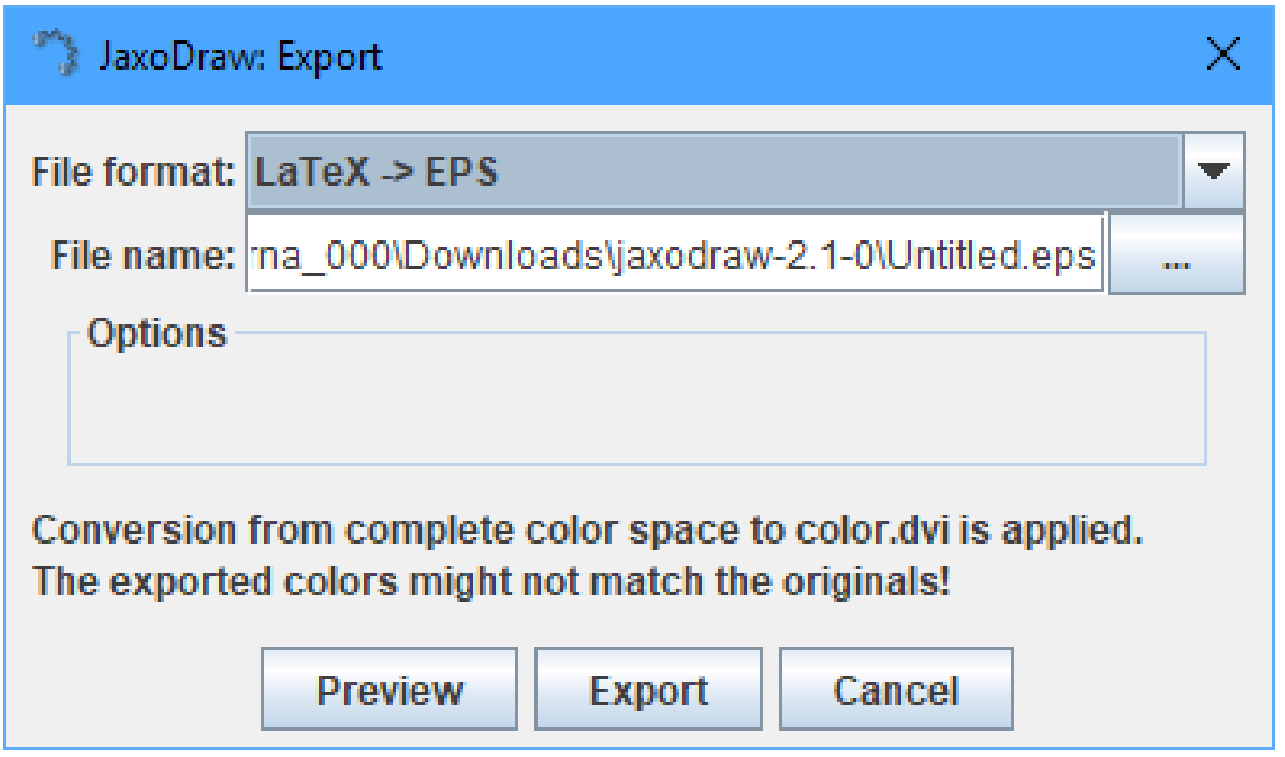


Figure 4. You can export your Feynman diagram in several different formats.

By default, JaxoDraw saves diagrams as XML files. Since it is a text file that essentially contains a description of the elements of your diagram, you theoretically could

manipulate a file with something as basic as a text editor. But, this isn't why you'd want to use JaxoDraw. If you click File→Export, a dialog window pops up where you can choose the details of how to export your diagram.

Since JaxoDraw is aimed at physicists, you'll most likely want to export your diagram into a form you easily can use with LaTeX, the most often used document layout system when theoretical physicists write for scientific journals. If this is the case, you actually have two options. You can save it as a pure LaTeX file that you can include in your document, or you can export as an encapsulated PostScript (EPS) file, which also can be manipulated within your document. If you're using something else to generate your document, you can export your diagram in some type of regular image format. Currently JPEG and PNG are supported, so you should be able to generate diagrams regardless of your needs with all of these options.

—Joey Bernard

News Briefs

Visit LinuxJournal.com for daily news briefs.

- Purism, maker of the security-focused Librem laptops, announced that it has partnered with Nitrokey to create Purekey, "Purism's own OpenPGP security token designed to integrate with its hardware and software. Purekey embodies Purism's mission to make security and cryptography accessible where its customers hold the keys to their own security." You can purchase a Purekey by itself or as an add-on with a laptop order. According to Purism's CSO Kyle Rankin, "By keeping your encryption keys on a Purekey instead of on a hard drive, your keys never leave the tamper-proof hardware. This not only makes your keys more secure from attackers, it makes using your keys on multiple devices more convenient."

- Vim 8.1 is now available. The major new feature of this release is that you now can run a terminal in a Vim window, which allows you to do things like run a command (like `make`) while editing in other windows or "use the new terminal debugger plugin for debugging inside Vim".

- AsteroidOS 1.0, the open-source operating system for smartwatches, is finally available after four years in the works. As posted on the AsteroidOS website, "AsteroidOS is built on standard Linux technologies including OpenEmbedded, opkg, Wayland, Qt5, systemd, BlueZ, and PulseAudio. This makes it the ideal platform to build any sort of wearable project you can imagine. Do you want to run Docker on your watch? AsteroidOS can do it. Do you want to run Quake on your watch? AsteroidOS can do that too. The sky is really the limit! Our community welcomes anyone interested in playing with a smartwatch project."

- It's here! Firefox 60 "Quantum" is available for download! Now with Client-Side Decorations (CSD) and much more!

- Mark Shuttleworth announced that Ubuntu 18.10 will be called Cosmic Cuttlefish. He also stressed that he is focusing on security, saying "If I had one big thing that I could feel great about doing, systematically, for everyone who

uses Ubuntu, it would be improving their confidence in the security of their systems and their data."

- There's a new automation framework called **Brigade**, which is written in Python. According to the **Networklore post**, "You could describe it as the automation framework for Pythonistas. This might strike you as something wonderful, or it could trigger your spider-sense. Writing code? Isn't that just for programmers?"

- Android P is finally addressing a privacy issue by restricting apps from monitoring your network activity (although this only affects apps that target Android P). xda **reported** recently that currently, "apps on Android can gain full access to the network activity on your device—even without asking for any sensitive permissions. These apps can't detect the content of your network calls, but they can sniff any outgoing or incoming connection via TCP/UDP to determine if you are connecting to a certain server."

- Mozilla **announce**d its new privacy-conscious approach to sponsored content. Earlier this year Mozilla began experimenting with showing a sponsored story occasionally in Pocket. The company is preparing to go live with it with the Firefox 60 release. Mozilla stresses that this new approach must not sacrifice user privacy: "All personalization happens on the client-side, without needing to vacuum up all of your personal data or sharing it with others." It also promises quality content, user control and transparency.

- After six years of development, a **new stable version of GIMP** has been released. Version 2.10.0 has a new default Dark theme and supports HIDPI displays and the GEGL image processing library. GIMP 2.10.0 also includes new tools, better file format support and an upgraded user interface, among other things. See the **release notes** for all the details.

Piventory: *LJ* Tech Editor's Personal Stash of Raspberry Pis and Other Single-Board Computers

Kyle Rankin is a Tech Editor and columnist at *Linux Journal* and the Chief Security Officer at Purism. He is the author of *Linux Hardening in Hostile Networks*, *DevOps Troubleshooting*, *The Official Ubuntu Server Book*, *Knoppix Hacks*, *Knoppix Pocket Reference*, *Linux Multimedia Hacks* and *Ubuntu Hacks*, and also a contributor to a number of other O'Reilly books. Rankin speaks frequently on security and open-source software including at BsidesLV, O'Reilly Security Conference, OSCON, SCALE, CactusCon, Linux World Expo and Penguicon. You can follow him at @kylerankin.

It's like an extra-geeky episode of Cribs featuring single-board computers.

By Kyle Rankin

I'm a big fan of DIY projects and think that there is a lot of value in doing something yourself instead of relying on some third party. I mow my own lawn, change my own oil and do most of my own home repairs, and because of my background in system administration, you'll find all sorts of DIY servers at my house too. In the old days, geeks like me would have stacks of loud power-hungry desktop computers around and use them to learn about Linux and networking, but these days, VMs and cloud services have taken their

place for most people. I still like running my own servers though, and thanks to the advent of these tiny, cheap computers like the Raspberry Pi series, I've been able to replace all of my home services with a lot of different small, cheap, low-power computers.

Occasionally, I'll hear people talk about how they have a Raspberry Pi or some other small computer lying around, but they haven't figured out quite what to do with it yet. And it always shocks me, because I have a house full of those small computers doing all sorts of things, so in this article, I describe my personal "Piventory"—an inventory of all of the little low-power computers that stay running around my house. So if you're struggling to figure out what to do with your own Raspberry Pi, maybe this article will give you some inspiration.

Primary NAS and Central Server

In "Papa's Got a Brand New NAS" I wrote about my search for a replacement for my rackmount server that acted as a Network-Attached Storage (NAS) for my house, along with a bunch of other services. Ultimately, I found that I could replace the whole thing with an ODroid XU4. Because of its octo-core ARM CPU, gigabit networking and high-speed USB3 port, I was able to move my hard drives over to a Mediasonic Probox USB3 disk array and set up a new low-power NAS that paid for itself in electricity costs.

In addition to a NAS, this server provides a number of backup services for my main server that sits in a data center. It acts as a backup mail server, authoritative DNS, and it also provides a VPN so I can connect to my home network from anywhere in the world—not bad for a little $75 ARM board.

Secondary NAS

Although my primary NAS runs fine, the four drive bays in my Mediasonic Probox limited my storage expansion options. I also didn't like the idea of the Odroid XU4 failing somehow and having my NAS be offline for days. So, I decided to set up a backup NAS with a duplicate USB3 disk array, with the idea that if either computer failed, I could move its disk array to the remaining machine. Although I liked the Odroid XU4, I had heard good things about the Espressobin board—in particular that it was able to get more performance out of its gigabit ports compared to other low-

Figure 1. Papa's New NAS

power computers. I ordered one, and although it was a bit more complicated to set up with its Armbian distribution compared to the Odroid XU4, when it did come on

the network, I noticed a significant improvement in its file transfer speeds.

The Espressobin isn't a perfect solution though. Unfortunately, so far I've noticed some stability issues with the Espressobin hardware—sometimes it just drops off the network, is unresponsive and requires a full power-cycle to bring it back online. I'm hoping that future firmware updates will help with the stability issues. If not, I may have to fall back to a second Odroid XU4.

Backup Server

In "Papa's Got a Brand New NAS", I also talked about my process of evaluating different boards to replace my old 1U home server. Before I decided on the ODroid XU4, I also considered the Banana Pi. It has a similar set of specifications and price as a Raspberry Pi of the same era with two important differences: onboard gigabit networking and a SATA2 port. When I decided to go with the ODroid XU4 for my NAS, I ended up with two Banana Pis sitting around with nothing to do. I decided to

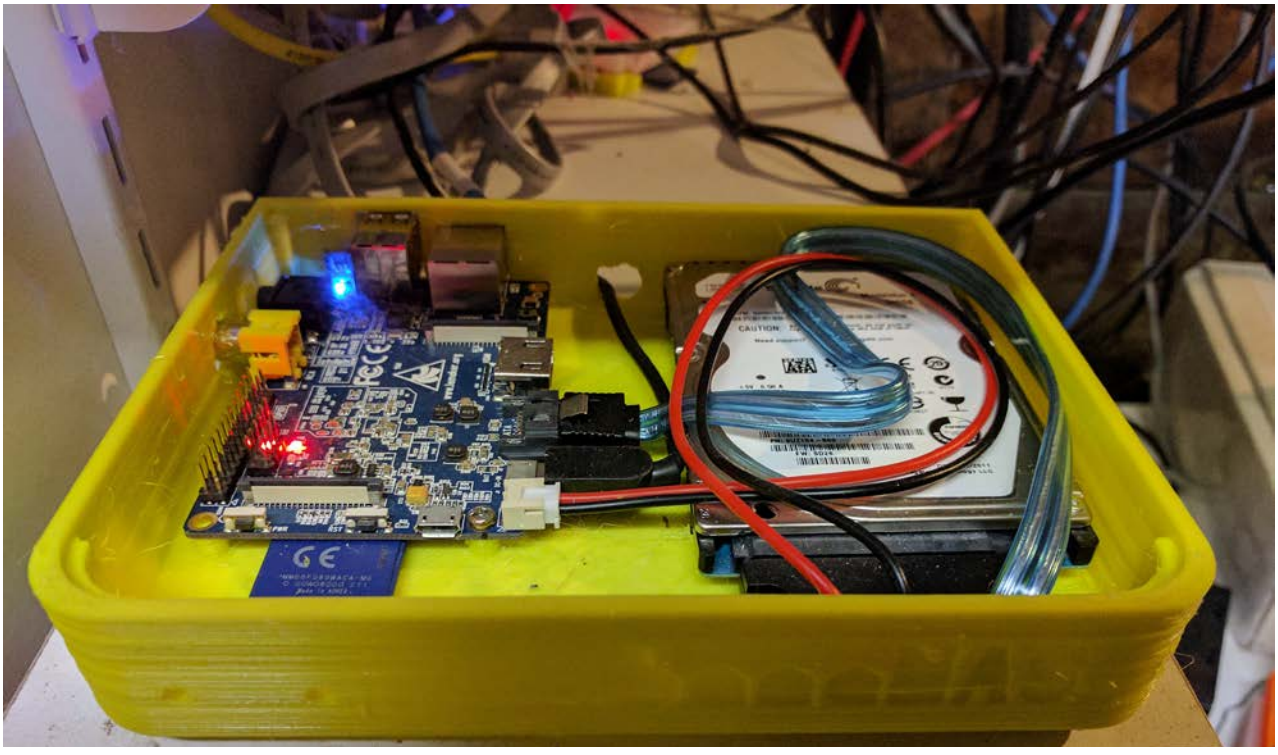


Figure 2. Banana Backup Server

put them to work, and the first one became my new BackupPC server.

In "Banana Backups", I elaborate on how I added a spare laptop hard drive to the Banana Pi's SATA2 port, used my 3D printer to print out a nice case for it, and then set up BackupPC, so I now have a completely standalone backup server that's keeping a separate copy of important files and settings in case I need to do a bare-metal restore of a different server.

Image Gallery

My BackupPC server accounts for one Banana Pi, but what about the other one? I didn't want it to go to waste, and around the time I was doing my primary NAS migration, I realized that the specific image gallery software I was using was no longer being maintained. What's worse, I couldn't even get the software to run on a modern distribution. When faced with this problem, some people would resort to a container, others a VM, and others probably would tell me to use a cloud service. What did I do? I installed an older but compatible Linux distribution on my spare Banana Pi so I could keep the same image gallery. Yes, from a security standpoint, this isn't ideal, but the server is pretty isolated, and I use strict access control to try to work around that risk.

Fermentation Fridge Controller

Along with my computer-related DIY projects, I also brew my own beer. I wanted to have better temperature control of my fermenting beer, but instead of buying one of those limited analog temperature controllers, I decided to build my own. In "Temper Pi", I describe the final iteration of that project when I migrate my temperature controller from a laptop to an original Raspberry Pi B. This system stays on all the time and uses a USB temperature probe and a series of X10 appliances to make sure that my fermentation fridge is the right temperature. If it's too hot, it turns the fridge on. If it's too cold, it turns the fridge off and turns on a small heating pad at the bottom of the fridge.

Media Center

I use my NAS to store music, videos, photos and other media. To play that media back requires—you guessed it—another Raspberry Pi. In this case, I use a Raspberry Pi 2 B running the OSMC distribution that provides a nice integrated Kodi system on top of

Raspbian. The media center computer is connected to an HDMI port on my TV and uses its gigabit port to access media from my central NAS. I just use a standard media center IR remote to control everything from the comfort of my couch.

RetroPie Retro Gaming System

All of my DIY projects don't leave that much time for gaming, but it's just as well, because I don't game much anyway, and the gaming I do tends to be retro gaming. In "Super Pi Brothers", I talked about the RetroPie Linux distribution—a distribution that provides a joystick-friendly interface to a bunch of different retro-gaming emulators. In my own home, I repurposed a Raspberry Pi B that used to be my original media center and 3D printed a NES-style case for it. I connected it to another HDMI port on my TV, bought a couple USB SNES joysticks for it, and whenever I do get that retro-gaming itch, I just switch TV inputs and go.

PiGrrl Zero

In addition to retro gaming at home, I like being able to bring my retro games with me when I travel. It's true that you can buy a lot of ready-made systems for this, but I decided to build my own. I used Adafruit's "PiGrrl Zero" kit that combines a Raspberry Pi Zero

Figure 3. PiGrrl Zero

along with a lot of electronics components, a battery and a small screen. You provide the 3D-printed case. At the end, you have a tiny RetroPie computer that fits in your pocket.

Octoprint 3D Printer Server

Another DIY hobby of mine involves 3D printing. I love the ability to make my own custom plastic objects at home, and I've made everything from toys for my son, replacement dishwasher parts, and even all of the cases for these various single-board computers lying around my house. I wrote a four-part series on 3D printing in Linux Journal called "What's New in 3D Printing", and in part four, I talk about the custom 3D printer control software called Octoprint. Octoprint includes a specific image just for Raspberry Pis, so I have another Raspberry Pi 2 B connected to my Printrbot Plus. The Octoprint server is connected to my 3D printer and also has a Raspberry Pi webcam aimed at the print bed. This means I can talk to the Octoprint Server over the network, send it print jobs and watch the results anywhere in my house (or if I connect to my VPN, anywhere with a network connection).

RV Media Center

The newest edition to my DIY server family is a new media center and "offsite" file backup server that sits in my RV running on a new Raspberry Pi 3 B+. (That server is the subject of a DIY Deep Dive article in this issue, so I won't elaborate on it much here.) In summary, it is mounted to the back of a 12V RV TV, runs OSMC and is connected to a large external hard drive. My home NAS keeps it synced up with a copy of all of my important personal and media files while it's in the driveway, so I not only have an extra off-site backup, I also have a copy of my favorite media for when I'm on the road.

Conclusion

Wow, I don't think I realized just how many single-board computers I have around the house until I wrote up this inventory. It's a good thing these are low-power computers, or my electricity bill would be crazy! If any of you have a single-board computer sitting around but you aren't sure what to do with it, I hope this article inspires you to put it to work. ■

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

Introducing PyInstaller

Reuven M. Lerner teaches Python, data science and Git to companies around the world. His free, weekly "better developers" email list reaches thousands of developers each week; subscribe here. Reuven lives with his wife and children in Modi'in, Israel.

Want to distribute Python programs to your Python-less clients? PyInstaller is the answer.

By Reuven M. Lerner

If you're used to working with a compiled language, the notion that you would need to have a programming language around, not just for development but also for running an application, seems a bit weird. Just because a program was written in C doesn't mean you need a C compiler in order to run it, right?

But of course, interpreted and byte-compiled languages do require the original language, or a version of it, in order to run. True, Java programs are compiled, but they're compiled into bytecodes then executed by the JVM. Similarly, .NET programs cannot run unless the CLR is present.

Even so, many of the students in my Python courses are surprised to discover that if you want to run a Python program, you need to have the Python language installed. If you're running Linux, this isn't a problem. Python has come with every distribution I've used since 1995. Sometimes the Python version isn't as modern as I'd like, but the notion of "this computer can't run Python programs" isn't something I've had to deal with very often.

However, not everyone runs Linux, and not everyone's computer has Python on it. What can you do about that? More specifically, what can you do when your clients don't have Python and aren't interested in installing it? Or what if you just want to write and distribute an application in Python, without bothering your users with additional installation requirements?

In this article, I discuss PyInstaller, a cross-platform tool that lets you take a Python program and distribute it to your users, such that they can treat it as a standalone app. I also discuss what it doesn't do, because many people who think about using PyInstaller don't fully understand what it does and doesn't do.

Running Python Code

Like Java and .NET, Python programs are compiled into bytecodes, high-level commands that don't correspond to the instructions of any actual computer, but that reference something known as a "virtual machine". There are a number of substantial differences between Java and Python though. Python doesn't have an explicit compilation phase; its bytecodes are pretty high level and connected to the Python language itself, and the compiler doesn't do that much in terms of optimization. The correspondence between Python source code and the resulting bytecodes is basically one-to-one; you won't find the bytecode compiler doing fancy things like inlining code or optimizing loops.

However, there's no doubt that Python runs bytecode, rather than your source code. You can see this in a number of different ways, the easiest of which is to create a Python module and then import that module. The module is translated into Python bytecodes and then saved to a file with a .pyc suffix. (In Python 3, this is under a directory called __pycache__, with separate byte-compiled versions for different Python versions and architectures.)

What does this all have to do with PyInstaller? Well, if you want to distribute a Python program, it's not enough to provide the byte-compiled output. You also need to provide a copy of Python, and that turns out to be a pain under certain circumstances, as I mentioned previously.

PyInstaller takes your Python code and byte-compiles it. But, then it also creates an executable application that basically loads Python and runs your program. In other words, each application you distribute with PyInstaller has a complete copy of Python within it, including the libraries needed to run your program. Normally, Python includes the entire standard library, but PyInstaller is smart enough to include only those modules it really needs, thus keeping the distribution size within reason.

Note that the copy of Python you have when using PyInstaller is used to create the distributable package. This means if you are running Python 3.4 on Linux, it's that copy of Python 3.4 for Linux that'll be included in your package. In other words, PyInstaller works across platforms, in that you can run it on Linux, Windows, macOS and other systems, but the resulting package is specifically for one architecture. It also means you need to be a bit careful when using PyInstaller on a computer that has multiple Python versions installed.

Installing PyInstaller

PyInstaller is most easily installed on a computer running Python with the standard `pip` command:

```
pip install -U --user pyinstaller
```

The `-U` flag indicates that you would like to upgrade PyInstaller, in case you already have installed it and the version on PyPI is newer. The `--user` flag indicates that you don't want to install it in the system's directories, but rather under your own home directory. Recently, I've become a fan of installing things with `--user`, largely because it avoids the need to think about permissions. However, it does mean that you need to add the "bin" directory from the `--user` location to your `PATH`.

If you're on a computer that has more than one Python version installed, it sometimes can be hard to know just which version is connected to `pip`. (Although `pip --version` will tell you which version of Python it's using.) For this reason, I sometimes do things the long way, as follows:

```
python3.6 -m pip install -U --user pyinstaller
```

The `-m` flag is sort of like the `import` statement in Python; running things in this way ensures that you're using the version you want.

Now that you've installed PyInstaller, let's use it to create a distributable Python application. I've created a new program called (very creatively) myapp.py. Here's the source code:

```
#!/usr/bin/env python3.6

import sys

print("Hello, and welcome to my app!")

print(f"We're running Python {sys.version}")

for i in range(10):
    print(f"{i} ** 2 = {i**2}, {i} ** 3 = {i**3}")
```

As you can see, this program imports the `sys` module, which provides access to the Python environment, as well as its variables and settings. I do this so that I can grab `sys.version` and ensure that the correct version is really running.

Next, I execute a "for" loop, for no reason other than that it gives me some output that I can see on the screen when the program runs.

In both cases, I use one of my favorite features from Python 3.6, `f-strings`, which allows me to interpolate expressions inside curly braces. This is, in my mind, far better than the previous ways this was done in Python, using the "%" operator on strings or (more recently) the `str.format` method.

So, let's assume you want to run this program on a colleague's machine. (Remember that your colleague needs to run the same operating system as you do, because the output from PyInstaller is going to be a binary based on the Python version you've

installed.) You can type:

```
pyinstaller myapp.py
```

And, you'll get a lot of output. I'm not going to review all of it, but here are some highlights:

```
468 INFO: PyInstaller: 3.3.1
468 INFO: Python: 3.6.3
470 INFO: Platform:
 ↪Linux-4.4.0-119-generic-x86_64-with-Ubuntu-16.04-xenial
475 INFO: wrote /root/myapp1/myapp.spec
```

The file "myapp.spec" describes the application you're creating with PyInstaller. You'll find that this file is created automatically when you run PyInstaller. Normally, PyInstaller is smart enough to figure out what files must be included in the resulting distribution, but in some cases, such as data files and shared libraries, you might have to edit the specfile and add them yourself:

```
491 INFO: Extending PYTHONPATH with paths
['/root/myapp1', '/root/myapp1']
```

When you say `import xyz` in Python, the language looks (for starters) in the current directory for "xyz.py". If it doesn't find that (or a bytecoded variation), it looks through the elements of `sys.path`, one by one, looking for "xyz.py". If you want to tell Python to look in some additional directories, you can set the `PYTHONPATH` environment variable. Here, PyInstaller is saying that it's modifying `PYTHONPATH` so that the program can find modules and packages defined in the current directory:

```
491 INFO: checking Analysis
```

PyInstaller analyzes your code in order to figure out which modules and packages you want to use. Used modules are included in the final distribution, while unused ones are ignored.

The "dist" Directory

There's a lot more to the output, but after running PyInstaller, you'll find that there's a "dist" directory, and that in that directory is another subdirectory with the name of your new application.

This directory contains your new Python application. Now, you can't just run it like that; it's still a bit more complex than your average executable. The idea is that you'll turn the directory into a zipfile, distribute the software to wherever you need it, unzip it on the destination machine, and then run the top-level program.

But what if you use a module that has not only a Python component, but also a compiled C component? PyInstaller handles that automatically. For example, say you're going to use NumPy in your program, how does PyInstaller handle the C portion, which is compiled?

In this case, PyInstaller noticed that you were using a module with a C component. And if you look in the "dist" directory, you'll now see a bunch of additional shared libraries (*.so files).

PyInstaller can't promise to work with all complex packages, but the authors have tried hard to provide a large degree of compatibility. For example, if you use the Cython package (for implementing Python modules in C or providing type hints), you'll find that PyInstaller handles it fine, including the appropriate files in the "dist" directory.

Conclusion

For years, many of my students and consulting clients have wanted to distribute Python code without needing to run the language itself. That's not possible, but PyInstaller does the next best thing, letting you distribute software in a fairly straightforward way. ■

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

The *LJ* Password Generator Tool

Dave Taylor has been hacking shell scripts on Unix and Linux systems for a really long time. He's the author of *Learning Unix for Mac OS X* and *Wicked Cool Shell Scripts*. He can be found on Twitter as @DaveTaylor and you can reach him through his tech Q&A site Ask Dave Taylor.

Mnemonic passwords generally stink. A random sequence of letters, digits and punctuation is more secure—just don't write down your passwords, like the knucklehead antagonist does in *Ready Player One*!

By Dave Taylor

In the password generating tool from my last article, at its most simple, you specify the number of characters you want in the password, and each is then chosen randomly from a pool of acceptable values. With the built-in `RANDOM` in the Linux shell, that's super easy to do:

```
okay="abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
okay="${okay}0123456789<>/?,>;:[{]}\|=+-_)(^%$#@!~
length=10
ltrs=${#okay}

while [ $length -ge 0 ]
do
   letter="${okay:$RANDOM % $ltrs:1}"
   result="$result$letter"
   length=$(( $length - 1 ))
done

echo "Result: $result"
```

In the actual script, I set `okay` to a single value rather than build it in two steps; this is just for formatting here online. Otherwise, `ltrs` is set to the length of `$okay` as a speedy shortcut, and the result is built up by using the string slicing syntax of:

```
${variable:indexlocation:length}
```

To extract just the fourth character of a string, for example, `${string:4:1}`, this works fine and is easy. The result speaks for itself:

```
$ sh lazy-passwords.sh
Result: Ojkr9>|}dMr
```

And, a few more:

```
Result: Mi8]TfJKVaH
Result: >MWvF2D/R?r
Result: h>J6\p4eNPH
Result: KixhCFZaesr
```

Where this becomes a more complex challenge is when you decide you don't want to have things randomly selected but instead want to weight the results so that you have more letters than digits, or no more than a few punctuation characters, even on a 15–20 character password.

Which is, of course, exactly what I've been building.

I have to admit that there's a certain lure to making something complex, if nothing else than just to see if it can be done and work properly.

Adding Weight to Letter Choices

As a result, the simple few lines above changed to this in my last article:

```
while [ ${#password} -lt $length ] ; do
```

```
   case $(( $RANDOM % 7 )) in
     0|1 ) letter=${uppers:$(( $RANDOM % ${#uppers})):1} ;;
     2|3 ) letter=${lowers:$(( $RANDOM % ${#lowers})):1} ;;
     4|5 ) letter=${punct:$(( $RANDOM % ${#punct}  )):1} ;;
     6 ) letter=${digits:$(( $RANDOM % ${#digits}  )):1} ;;
   esac
  password="${password}$letter"
done
```

In the above code, I'm assigning probability that a given letter will be one of four classes: uppercase, lowercase, punctuation or digits. The first three are each 2:7 chances, and punctuation is 1:7, or half as likely to be produced.

A run of four iterations of the above algorithm produces these results:

```
q.x0bAPmZb
P}aWX2N-U]
5jdI&ep7rt
-k:TA[1I!3
```

Since random is, well, random, in both situations, you actually can end up with a password that includes no punctuation. So, how do you force a specific number of occurrences of punctuation symbols? One solution is to check after the fact to see if you met your target count.

To do that, you'll need two things: goals and counters. Let's add the former as startup options in a typical `getops` block:

```
while getopts "l:d:p:" arg
do
   case "$arg" in
     l) length=$OPTARG ;;
     d) digitGoal=$OPTARG ;;
```

```
      p) punctGoal=$OPTARG ;;
      *) echo "Valid -l length, -d digits, -p punctuation"
         exit 1 ;;
    esac
done
```

The counters are also easy; every time a digit or punctuation condition is met in the case statement (shown earlier), you increment the counter by one.

Finally, at the end, you can compare the two and see if you met your weighted randomization goals:

```
if [ $digitsAdded -lt $digitGoal ] ; then
  echo "Didn't add enough digits. [goal = $digitGoal,
    inserted = $digitsAdded]"
  exit 1
elif [ $punctAdded -lt $punctGoal ] ; then
  echo "Didn't add enough punctuation. [goal = $punctGoal,
    inserted = $punctAdded]"
  exit 1
fi
```

If the total length of the requested password is reasonable compared to the random chance that a digit or punctuation character will be added, this will work fine. A 15-character password with at least two punctuation characters will be generated without a hiccup almost every single time.

Although once in a while:

```
$ sh makepw.sh -l 15 -p 4
Didn't add enough punctuation. [goal = 4, inserted = 1]
```

This begs the most important question of the script algorithm: what do you do once

you realize that you haven't met your digit and punctuation character goals?

Failed Password Generation: Now What?

One solution is simply to try again, but if the user sets up an impossible situation, like a six-character password with four digits and four punctuation characters, or even four and two, that's *no bueno*.

Another possibility is to step through the generated password, replacing unconstrained values (such as upper and lowercase) with the specific value required. This has the consequence that if you ask for a lot of punctuation or digits, you're going to end up having those requested characters front-loaded, which isn't exactly random.

So, Let's Rethink the Problem

What if, instead of producing a random password, you split it into two steps? The first step is to generate the required number of random digits and random punctuation characters, add completely random values to add up to the desired length, then "shuffle" the result to produce the final password.

I know, you're almost done with this program, but that's a really interesting solution that sidesteps a lot of problems, so let's just retrench and start over!

Actually, it's not that bad, because most of the work's already been done. This will just make it simpler:

```
while [ ${#password} -lt $length ] ; do
   if [ $digitsAdded -lt $digitGoal ] ; then
     letter=${digits:$(( $RANDOM % ${#digits} )):1}
     digitsAdded=$(( $digitsAdded +1 ))
   elif [ $punctAdded -lt $punctGoal ] ; then
     letter=${punct:$(( $RANDOM % ${#punct} )):1}
     punctAdded=$(( $punctAdded +1 ))
   else
     case $(( $RANDOM % 7 )) in
```

```
        0|1) letter=${uppers:$(($RANDOM % ${#uppers})):1} ;;
        2|3) letter=${lowers:$(($RANDOM % ${#lowers})):1} ;;
        4|5) letter=${punct:$(($RANDOM % ${#punct} )):1}
             punctAdded=$(( $punctAdded + 1 ))        ;;
        6) letter=${digits:$(( $RANDOM % ${#digits} )):1}
             digitsAdded=$(( $digitsAdded +1 ))       ;;
      esac
    fi
    password="${password}$letter"
done
```

Without the final password-scrambler code, here's what you get with a couple invocations for a 15-character password with at least four digits and at least four punctuation characters:

```
$ sh makepw.sh -l 15 -p 4 -d 4
Interim password generated: 8119?:)@_g&rw%=
$ sh makepw.sh -l 15 -p 4 -d 4
Interim password generated: 7599}(|&l*4KFY/
```

You clearly can see how these are front-loaded, with the digits required, the punctuation required and then "everything else".

There are a lot of ways to shuffle the letters in a word within a shell script, ranging from invoking Perl or Awk to using the Linux `shuf` command to solving it yourself. I'm going to leave this as an exercise for the reader, because with that small added step, you've got a fully functional password generator that's ready to take on your hundreds of system users. ■

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

What's New in Kernel Development

By Zack Brown

Zack Brown is a tech journalist at *Linux Journal* and *Linux Magazine*, and is a former author of the "Kernel Traffic" weekly newsletter and the "Learn Plover" stenographic typing tutorials. He first installed Slackware Linux in 1993 on his 386 with 8 megs of RAM and had his mind permanently blown by the Open Source community. He is the inventor of the *Crumble* pure strategy board game, which you can make yourself with a few pieces of cardboard. He also enjoys writing fiction, attempting animation, reforming Labanotation, designing and sewing his own clothes, learning French and spending time with friends'n'family.

New NOVA Filesystem

Andiry Xu (working with **Lu Zhang**, **Steven Swanson** and others) posted patches for a new filesystem called **NOVA** (NOn-Volatile memory Accelerated). Normal **RAM** chips are wiped every time you turn off your computer. Non-volatile RAM retains its data across reboots. Their project targeted byte-addressable non-volatile memory chips, such as **Intel**'s **3DXpoint DIMMs**. Andiry said that the current incarnation of their code was able to do a lot already, but they still had a big to-do list, and they wanted feedback from the kernel people.

Theodore Y. Ts'o gave the patches a try, but he found that they wouldn't even compile without some fixes, which he posted in reply. Andiry said they'd adapt those fixes into their patches.

The last time NOVA made an appearance on the kernel mailing list was August 2017, when Steven made a similar announcement. This time around, they posted a lot more patches, including support for **SysFS** controls, **Kconfig** compilation options and a significant amount of documentation.

One of NOVA's main claims to fame, aside from supporting non-volatile RAM, is that it is a log-based filesystem. Other filesystems generally map out their data structures on disk and update those structures in place. This is good for saving seek-time on optical and magnetic disks. Log-based filesystems write everything sequentially, trailing old data behind them. The old data then can be treated as a snapshot of earlier states of the filesystem, or it can be reclaimed when space gets tight.

Log-based filesystems are not necessarily preferred for optical and magnetic drives, because sequential writes will tend to fragment data and slow things down. Non-volatile RAM is based on different technology that has faster seek-times, making a log-based approach a natural choice.

NOVA goes further than most log-based filesystems, which tend to have a single log for the whole filesystem, and instead maintains a separate log for each inode. Using the log data, NOVA can perform writes either in place like traditional filesystems or as copy-on-write (COW) operations, which keep the old version of a file until the new version has been written. This has the benefit of being able to survive catastrophic events like sudden power failures in the middle of doing a write, without corrupting the filesystem.

There were lots of responses to the patches from Andiry and the rest of his team. Most were bug reports and criticism, but no controversy. Everyone seemed to be interested in helping them get their code right so the patches could get into the main tree quickly.

Removing All Syscall Invocations from Kernel Space

There's an effort underway to reduce and ultimately remove all system call invocations from within kernel space. **Dominik Brodowski** was leading this effort, and he posted some patches to remove a lot of instances from the kernel. Among other things, he said, these patches would make it easier to clean up and optimize the **syscall** entry points, and also easier to clean up the parts of the kernel that still needed to pretend to be in userspace, just so they could keep using syscalls.

The rationale behind these patches, as expressed by **Andy Lutomirski**, ultimately was to prevent user code from ever gaining access to kernel memory. Sharing syscalls between kernel space and user space made that impossible at the moment. Andy hoped the patches would go into the kernel quickly, without needing to wait for further cleanup.

Linus Torvalds had absolutely no criticism of these patches, and he indicated that this was a well desired change. He offered to do a little extra housekeeping himself with the kernel release schedule to make Dominik's tasks easier. Linus also agreed with Andy that any cleanup effort could wait—he didn't mind accepting ugly patches to update the syscall calling conventions first, and then accept the cleanup patches later.

Ingo Molnar predicted that with Dominik's changes, the size of the compiled kernel would decrease—always a good thing. But Dominik said no, and in fact, he ran some quick numbers for Ingo and found that with his patches, the compiled kernel was actually a few bytes larger. Ingo was surprised but not mortified, saying the slight size increase would not be a showstopper.

This project is similar—although maybe smaller in scope—to the effort to get rid of the **big kernel lock** (BKL). In the case of the BKL, no one could figure out for years even how to begin to replace it, until finally folks decided to convert all BKL instances into identical local implementations that could be replaced piecemeal with more specialized and less heavyweight locks. After that, it was just a question of slogging through each one until finally even the most finicky instances were replaced with more specialized locking code.

Dominik seems to be using a similar technique now, in which areas of the kernel that still need syscalls can masquerade as userspace, while areas of the kernel that are easier to fix get cleaned up first.

Loading Arbitrary Executables as Kernel Modules

Alexei Starovoitov posted some patches to allow the kernel to load regular **ELF** binaries (aka plain executables) as kernel modules. These modules would

be able to run user-mode helper routines instead of being absolutely confined to kernel space.

Alexei listed a variety of benefits for this. For one thing, as a user process, an ELF-based module could crash without bringing down the rest of the kernel. And although the ELF modules would run with root privileges, he said that a security breach would not lead directly into accessing the kernel's inner workings, but at least initially would be confined to userspace. The ELF module also could be terminated by the out-of-memory (OOM) killer, in case of need, or ended directly by a human administrator. It additionally would be feasible to subject ELF-based modules to regular userspace debugging and profiling, using the vast array of tools available for that.

Initially there were various technical questions and criticisms, but no one spoke out immediately against it. Linus Torvalds said he liked the feature, but he wanted one change: to make the type of module visible in the system logs. He said:

> When we load a regular module, at least it shows in lsmod afterwards, although I have a few times wanted to really see module load as an event in the logs too. When we load a module that just executes a user program, and there is no sign of it in the module list, I think we *really* need to make that event show to the admin some way.

And he said specifically, "I do *not* want this to be a magical way to hide things."

Andy Lutomirski raised a pertinent question: why not just retool the **modprobe** program to handle ELF binaries as desired, rather than doing anything with kernel code at all? In other words, why couldn't this feature be implemented entirely outside the kernel?

But Linus replied:

> The less we have to mess with user-mode tooling, the better.

> We've been *so* much better off moving most of the module loading logic to the kernel, we should not go back in the old broken direction.
>
> I do *not* want the kmod project that is then taken over by systemd, and breaks it the same way they broke firmware loading.
>
> Keep modprobe doing one thing, and one thing only: track dependencies and mindlessly just load the modules. Do *not* ask for it to do anything else.
>
> Right now kmod is a nice simple project. Lots of testsuite stuff, and a very clear goal. Let's keep kmod doing one thing, and not even have to care about internal kernel decisions like "oh, this module might not be a module, but an executable".
>
> If anything, I think we want to keep our options open, in the case we need or want to ever consider short-circuiting things and allowing direct loading of the simple cases and bypassing modprobe entirely.

At one point, **Kees Cook** did offer some more serious criticisms of the patch's basic goal. Primarily, he noticed that Alexei's patch could be used to —and was intentionally designed to —execute arbitrary code in userspace automatically. This was different from the kernel's normal approach to modules, in which they could be loaded but not executed automatically.

Kees said, "This just extends all the problems we've had with defining security boundaries with modules out to umh [user-mode helper code] too. I would need some major convincing that this can be made safe."

He pointed out that a certain class of kernel bugs—apparently prevalent in the recent past—could redirect module loading outside of a virtual machine (that is, a container) and into the main kernel itself. And since containers could trigger loading an arbitrary module, this meant that a hostile user potentially could load an ELF module, redirect it back to the main kernel, and execute its attacking code immediately with full privileges.

Kees refused to let the patch go into the kernel as written. He said:

> At the very least, you need to solve the execution environment problems here: the ELF should run with no greater privileges than what loaded the module, and very importantly, must not be allowed to bypass these checks through autoloading. What *triggered* the autoload must be the environment, not the "modprobe", since that's running with full privileges.

On the flip side, however, Kees acknowledged that Alexei's patch was an "interesting idea. I think it *can* work, it just needs much much more careful security boundaries and to solve our autoloading exposures too."

However, Alexei characterized Kees' response as "security paranoia without single concrete example of a security issue."

And Andy also disagreed with Kees' assessment. He pointed out that Kees' issue depended on an attacker finding and exploiting an additional vulnerability that would allow containers to redirect a module outside of itself—something that was not a kernel feature and that would be treated as a bug if it were ever discovered.

Kees agreed with Andy that the problem was not with Alexei's code but instead with potential vulnerabilities elsewhere in the kernel. He said, "I just don't want to extend that problem further." And he added that he wasn't opposed to Alexei's patch, but that his concerns were not paranoia, and "there are very real security boundary violations in this model."

At one point, in defense of Alexei's approach, Andy said, "I don't see how this is any more exploitable than any other init_module()." And Linus replied:

> Absolutely. If Kees doesn't trust the files to be loaded, an executable—even if it's running with root privileges and in the initns—is still fundamentally weaker than a kernel module.
>
> So I don't understand the security argument AT ALL. It's nonsensical. The executable

> loading does all the same security checks that the module loading does, including the signing check.

Kees acknowledged that his concern was not with Alexei's code itself, or even with the design of the feature. But he felt that if certain other bugs did appear in the kernel—as they had before—then someone would be able to exploit the feature to run arbitrary code at the root level.

However, Linus had spoken, and Kees' concern over potential future bugs were apparently not a showstopper. And just to hammer it home, **David S. Miller** reiterated Linus' point that kernel modules were far more dangerous than executable code, because they could access any container and namespace they pleased.

But this was not the end of the story!

Close by to this part of the conversation, Linus said to Kees:

> My own personal worry is actually different—we do check the signature of the file we're loading, but we're then passing it off to execve() not as the image we loaded, but as the file pointer. So the execve() will end up not using the actual buffer we checked the signature on, but instead just re-reading the file.

Among other things, Linus said, "somebody could maybe try to time it and modify the file after-the-fact of the signature check, and then we execute something else."

He went on to say:

> Initially, I thought it was a non-issue, because anybody who controls the module subdirectory enough to rewrite files would be in a position to just execute the file itself directly instead. But it turns out that isn't needed. Some bad actor could just do `finit_module()` with a file that they just *copied* from the module directory.

Linus said this issue had to be addressed before the patch could go into the kernel.

Andy also noticed something else that might be a deal-killer. He said:

> This patch is a potentially severe ABI break. Right now, loading a module *copies* it into memory and does not hold a reference to the underlying fs. With the patch applied, all kinds of use cases can break in gnarly ways. Initramfs is maybe okay, but initrd may be screwed. If you load an ET_EXEC module from initrd, then umount it, then clear the ramdisk, something will go horribly wrong. Exactly what goes wrong depends on whether userspace notices that umount() failed. Similarly, if you load one of these modules over a network and then lose your connection, you have a problem.

He explained further, "Without your patch, init_module doesn't keep using the file, so it's common practice to load a module and then delete or unmount it. With your patch, the unmount case breaks. This is likely to break existing userspace, so, in Linux speak, it's an ABI break."

At this point—regarding Linus' security exploit—Andy felt that Kees' thumbs-up would be more important than he had at first. Kees' responsibility was module security, which Andy had thought was not an issue earlier in the discussion. Now that it was, it had become more important to get Kees' blessing on this patch. Andy pointed out to Alexei, "Kees is very reasonable, and he'll change his mind and ack a patch that he's nacked when presented with a valid technical argument."

However, he also said:

> My ABI break observation is also a major problem, and Linus is going to be pissed if this thing lands in his tree and breaks systems due to an issue that was raised during review. So I think you need to either rework the patch or do a serious survey of how all the distros deal with modules (dracut, initramfs-tools, all the older stuff, and probably more) and make sure they can all handle your patch.

Alexei replied that neither of these problems were real issues. The ABI (application binary interface) break didn't really break the kernel ABI, and

the security issue was not a real concern. He said, “I think you need to stop overreacting on a non-issue.”

There was a bit of back and forth between them. It turned out that Alexei didn’t believe there was an ABI breakage because in his intended use case, everything would be done identically to the way it was now, and so nothing would be broken. But **Greg Kroah-Hartman** replied, “For *your* use case, yes. For mine and Andy’s and someone else’s in the future, it might be.” He added, “You are creating a very generic, new, user/kernel api that a whole bunch of people are going to want to use. Let’s not hamper the ability for us all to use this right from the beginning please.”

And in terms of specific use cases, Greg said:

> We have userspace drivers for USB today, being able to drag that out-of-tree codebase into the kernel is a *HUGE* bonus, and something that I would love to do for a lot of reasons. I also can see moving some of our existing in-kernel drivers out of the kernel in a way that provides “it just works” functionality by using this type of feature.

A bunch of folks, including Linus, started debating ways to address the problems that had been identified so far. Alexei got on board after a while and started implementing changes as they were identified by the group.

It seems clear that this feature will go into the kernel. It provides cool functionality that is hotly desired by Linus and others. But the timing of getting the code into the kernel will depend on how well Alexei fixes the various problems, and whether new security or ABI issues arise.

This whole discussion was interesting on a number of levels. I particularly like the speed with which the critics and defenders of Alexei’s patch would change positions, without regard to ego or fear of being seen as “wrong” or anything like that. And what had started as a wholehearted acceptance of a new feature,

became concern over its possible problems and a quest to resolve them in a useful way.

I also like that Alexei's initial minor defensiveness was not treated as cause for bullying, and everyone simply tried to keep the discussion productive. It doesn't always go that way in kernel development.

And of course, I also find it exciting to be able to look in on discussions of potential kernel weaknesses, how they might be exploited by hostile actors, and what might be done to stop them. In the early days, that was exactly how the kernel folks used to talk about Microsoft—right out in the open! What might Microsoft do to destroy open source? How might it destroy the GPL? How might it destroy Linux? And all the while, the kernel people knew that the Microsoft people were reading their every post, just as they know now that hostile attackers are eagerly poking and prodding the mailing list discussions and every patch submission, looking for usable exploits.

Removing Support for Dead Hardware

Arnd Bergmann submitted a patch to remove the Linux ports for a variety of architectures, including **blackfin**, **cris**, **frv**, **m32r**, **metag**, **mn10300**, **score** and **tile**. To do this, he worked directly with the former maintainers of each port to make sure the code removal was done right and didn't break anything in the mainline kernel or anywhere else.

The bottom line was that no one used those architectures anymore. He offered his analysis of why this had happened, saying:

> It seems that while the eight architectures are extremely different, they all suffered the same fate: There was one company in charge of an SoC line, a CPU microarchitecture and a software ecosystem, which was more costly than licensing newer off-the-shelf CPU cores from a third party (typically ARM, MIPS, or RISC-V). It seems that all the SoC product lines are still around, but have not used the custom CPU architectures for several years at this point. In contrast,

> CPU instruction sets that remain popular and have actively maintained kernel ports tend to all be used across multiple licensees.

Linus Torvalds had no objection to ripping those architectures out of the kernel, but he did say, "I'd like to see that each architecture removal is independent of the others, so that if somebody wants to resurrect any particular architecture, he/she can do so with a revert."

Linus pulled the patch into the main kernel tree and noted with glee that it took a half-million lines of code out of the kernel.

Linus was not the only one who wanted to ensure the possibility of easily resurrecting those architectures. **Geert Uytterhoeven** wanted to know exactly what would be required, since he had an interest in the formerly removed and later resurrected **arch/h8300** architecture, currently still in the kernel and going strong. And he pointed out, "In reality, a resurrection may not be implemented as a pure revert, but as the addition of a new architecture, implemented using modern features."

To which **Pavel Machek** complained, "By insisting on new features instead of pure revert + incremental updates, you pretty much make sure resurrection will not be possible."

But Arnd pointed out, "now that the other architectures are gone, a lot of changes can be done more easily that will be incompatible with a pure revert, so the more time passes, the harder it will get to do that."

And he added, "Some of the architectures (e.g. tile or cris) have been kept up to date, but others had already bitrotted to the point where they were unlikely to work on any real hardware for many releases, but a revert could still be used as a starting point in theory."

In any case, Arnd's patches are sailing into the kernel, and those architectures

are, for the moment, dead. But even so, it's odd to see them removed. There will certainly come a day in the far distant future when a hardware aficionado decides to write an emulator for them; and then they'll have to root around in the ancient logs of the kernel git repository to find versions of Linux that supported them, and those versions of Linux will run only on ancient hardware that itself will exist only in emulated form. So they'd have ancient hardware running on an ancient kernel on top of an emulated ancient CPU being simulated by the latest Quantum Sine-Qua-Non Ultra Linux (QSQNUL) , powered by black-hole turbines and puppy-dog hearts.

It'll happen.

Note: if you're mentioned in this article and want to send a response, please send a message with your response text to ljeditor@linuxjournal.com *and we'll run it in the next Letters section and post it on the website as an addendum to the original article.* ■

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.


DEEP
DIVE
DIY

Why You Should Do It Yourself

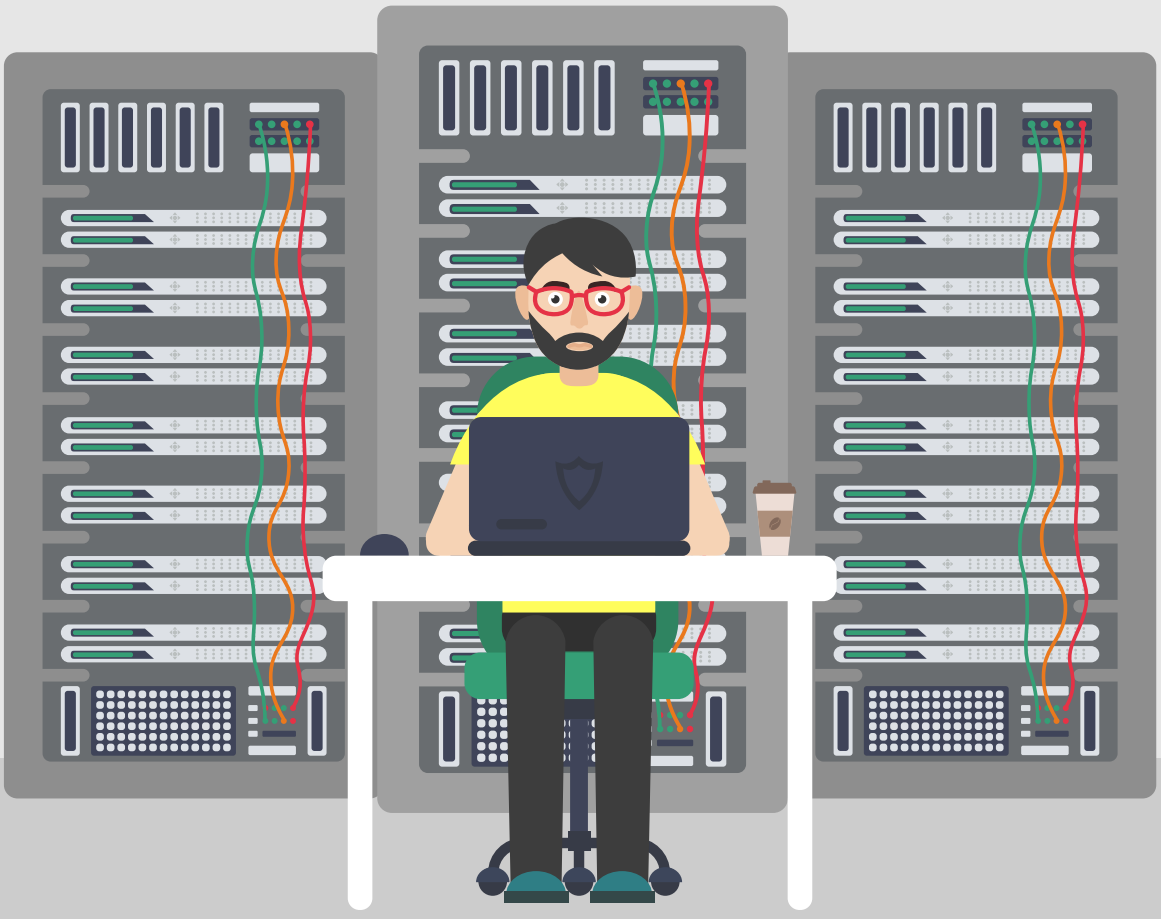
Bring back the DIY movement and start with your own Linux servers.

By Kyle Rankin

It wasn't very long ago that we lived in a society where it was a given that average people would do things themselves. There was a built-in assumption that you would perform basic repairs on household items, do general maintenance and repairs on your car, mow your lawn, cook your food and patch your clothes. The items around you reflected this assumption with visible and easy-to-access screws, spare buttons sewn on the bottom of shirts and user-replaceable parts.

Through the years though, culture has changed toward one more focused on convenience. The microeconomic idea of "opportunity cost" (an idea that you can assign value to each course of action and weigh it against alternative actions you didn't take) has resulted in many people who earn a reasonable wage concluding that they should do almost nothing themselves.

The typical thinking goes like this: if my hourly wage is higher than the hourly cost of a landscaping service, even though that landscaping service costs me money, it's still *cheaper* than if I mowed my own lawn, because I could somehow be earning my hourly wage doing something else. This same calculation ends up justifying oil-change and landscaping services, microwave TV dinners and replacing items when they break instead of repairing them yourself. The result has been a switch to a service-oriented economy, with the advent of cheaper, more disposable items that hide their screws and vehicles that are all but hermetically sealed under the hood.



This same convenience culture has found its way into technology, with entrepreneurs in Silicon Valley wracking their brains to think of some new service they could invent to do some new task for you. Linux and the Open Source movement overall is one of the few places where you can still find this do-it-yourself ethos in place.

When referring to proprietary software, Linux users used to say "You wouldn't buy a car with the hood welded shut!" With Linux, you can poke under the hood and see exactly how the system is running. The metaphorical screws are exposed, and you can take the software apart and repair it yourself if you are so inclined. Yet to be honest, so many people these days *would* buy a car with the hood welded shut. They also are fine with buying computers and software that are metaphorically welded shut all

justified by convenience and opportunity cost.

Yet there is something missing in most of these opportunity cost calculations—the less tangible benefit you get from increased knowledge and skills when you figure out how things work and how to repair them, and the satisfaction and pride you feel when you see the results of your own efforts. When you do things yourself, the mystique starts to fade away, and you realize that most of the things around you aren't nearly as complex or magical or inaccessible as you once thought they were. The increased confidence you got when repairing your lamp might encourage you to figure out how to replace that faulty light switch in your house.

With the advent of cloud services, it's incredibly easy to have someone else set up Linux and open-source services for you, and when they break, throw them away and start from scratch. However, this convenience comes with a cost of its own, and not just the hourly rate for your servers. These services and abstraction layers make things easier to use but harder to understand. It adds a mystique to technology and makes it seem much more complex than it really is under the hood. It also makes it easy to avoid learning how things actually work and hard to learn how to fix them.

When I started out with Linux on the desktop, I essentially got an education in software development, networking and system administration as part of just making Linux work. Using Linux as a server platform immersed me in computer fundamentals even more. I learned more about Linux by breaking something and then fixing it myself than I ever would have in a certification class or by handing it off to tech support. I learned more about networking fundamentals by physically connecting two Linux servers and trying to get them to talk to each other than I ever would by clicking a few buttons in a cloud dashboard.

These days Linux has become a lot easier to use, but along with that ease, we've lost most of those built-in learning opportunities. How do new junior engineers learn how networking really works these days? How do they learn how to fix a broken Linux server? I want to encourage you to search for these learning opportunities. Learn what's going on under the hood. Get a spare Raspberry Pi and set up your own web, DHCP, DNS and email servers from scratch.

Eventually that server will break, and if you don't give up, with a bit of research and troubleshooting, you'll be able to fix it all yourself. In the process, you'll find this stuff really isn't as complex as you thought it was, and you'll find yourself wondering: what *else* can I do myself?

You gain something else when you do it yourself—independence, freedom and control. One of the real risks with proprietary software is the fact that control (and freedom) is taken away from you, as you are beholden to the software vendor to patch bugs and add features. There are countless examples of developers who got their start in open-source software because they wanted to add one extra feature to an existing program, fix one annoying bug or get some random piece of hardware working. Because the source code was available, they could freely inspect it, learn how the software worked and figure out how to change it themselves.

So, next time that cloud server is spitting out 500 errors and you have no idea why, resist the temptation to kill it and spawn a new one. Take a minute to poke under

the hood and see if you can figure out the problem yourself. The next time you need some new service around the house, resist the urge to buy some shiny IoT gadget. See if you can make it yourself with a Raspberry Pi or Arduino and a little bit of software. After all, that's what most of these IoT devices are underneath their shiny exteriors. Sure it may take a little more time, but when your neighbor's IoT appliance starts DDoS-ing the internet at large, you'll have the skills to help troubleshoot it. ■

Kyle Rankin is a Tech Editor and columnist at *Linux Journal* and the Chief Security Officer at Purism. He is the author of *Linux Hardening in Hostile Networks, DevOps Troubleshooting, The Official Ubuntu Server Book, Knoppix Hacks, Knoppix Pocket Reference, Linux Multimedia Hacks* and *Ubuntu Hacks*, and also a contributor to a number of other O'Reilly books. Rankin speaks frequently on security and open-source software including at BsidesLV, O'Reilly Security Conference, OSCON, SCALE, CactusCon, Linux World Expo and Penguicon. You can follow him at @kylerankin.

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.

PHOTO CREDIT: JOY RANKIN

DIY: RV Offsite Backup and Media Server

What better way to add a geeky touch to #vanlife than with a Linux server in your RV?

By Kyle Rankin

One easily could make the strong argument that an RV is the ultimate DIY project playground. It combines all of the DIY projects you could perform on a vehicle with the DIY projects for a home. Add to that the fact that you may spend days living in a small house on wheels navigating highways, forests and deserts, and you have a whole other class of DIY projects to make the most of that smaller space. RVs also offer a whole suite of power options from 12V deep cycle batteries to 110V shore power to generators and alternators to solar power, so there's a whole class of electrical DIY projects related to making the most of your changing power options.

And if you're a geek, having an RV introduces a whole other level of DIY possibilities. First, there are all of the electronics projects to manage switching between power sources, tracking energy consumption and keeping those batteries charged. Then there's an entire category of projects related to internet access while away from home that involve everything from mobile WiFi hotspots to cellular-boosting networks to roving satellite internet (and if you're clever, a smart router that routes you to the best and cheapest available option). Finally, there are several project possibilities related to the computer systems in the RV, including local switches and routers, personal computers that turn the RV into a mobile office, and media centers so you can watch TV and movies from the road.

It just so happens that I recently got an RV—a 1996 Roadtrek 170 to be exact.

Figure 1. Introducing "Van Winkle" (Photo Credit: Joy Rankin)

Although this purchase has spawned a huge list of DIY projects, my very first Linux-based project focuses on the media center. At home, my media center is a Raspberry Pi running OSMC, and it works great for accessing my ripped DVDs and CDs from my NAS and playing them on my living-room TV. When I got the RV, I realized that one of the first things we'd want is a way to access all of that media from the road, even if we were in the middle of the woods.

In this article, I describe all the steps I took to build a media server just for the RV that maintains an up-to-date copy of my media and even syncs up automatically when it's parked in my driveway. It turns out that in the process of building a media server, I ended up with a pretty great off-site backup solution as well. Even if you don't own an

RV, you could adapt these steps to add your own semi-offsite backup to your car.

Bill of Materials

Before I go through each step of this project, I want to post the complete list of hardware I used. I try to explain why I chose the hardware I did below, and depending on your needs, some of this hardware might be optional. For instance, if you have good WiFi reception from your vehicle, you may not need the WiFi extender I bought. If you just want to build an offsite file backup solution without any media playback, you wouldn't need to buy the TV or mounting brackets.

- AXESS TV1705-15 15" LED HDTV: $124.99
- Mount-It! Lockable RV TV Mount: $43.99
- Raspberry Pi 3 B+: $45.99
- Western Digital 8TB MyBook USB3 Hard Drive: $170
- Two-pack 16GB SanDisk SDHC MicroSD card: $10.78
- One-foot HDMI cable: $7.99
- Windows 7 Vista XP Media Center MCE PC Remote Control and Infrared Receiver: $16.90
- VONETS VAP11G-300 WiFi Repeater: $19.89
- **Total: $440.53**

Step 1: Media Server Hardware

The first step in this process was selecting the hardware and software for the media server itself. Since the hardware was going to run from my RV "house battery", the lower-power the better. I had planned on using some form of

pre-made distribution for the Kodi media center software (I was torn between OpenELEC, LibreELEC and OSMC), so I wanted to choose hardware that was well supported by those distributions. Since the computer was going to perform media playback, and all of these low-power devices have relatively slow general-purpose ARM CPUs in them, it also was important that the distributions on my list could take advantage of hardware-accelerated playback for common media codecs. I wanted to sync my files over to this hardware wirelessly, so it was also important that the hardware have a wireless card. Finally, because I planned on attaching a large external USB3 hard drive to the computer to store all of my media, ideally the computer would have a higher-speed USB3 port.

As I evaluated hardware, the first major change I made was ditching my USB3 requirement in favor of hardware-accelerated media playback. Most of the single-board computers I evaluated that had USB3 support (like the Odroid XU4 I used for my NAS) didn't have guaranteed hardware acceleration for media. As I did research on the issue though, I realized that while USB2 bandwidth is relatively low, it turns out that many people in the community stream large media over USB2 and that it should be fine for my uses. With media playback out of the way, my other concern was in syncing files from my home network, but in that case, my WiFi connection was the real bottleneck.

Now that I no longer had to worry about USB3 support, I started to limit my choices to hardware explicitly supported by LibreELEC, as it was the front-runner in my distribution selection. For my final hardware selection, I narrowed it down to the Raspberry Pi 3 B+ or an Odroid C2—a competitor to the Raspberry Pi series that promised great graphics performance and had an integrated IR sensor. In the end I settled on the Raspberry Pi 3 B+ for the following reasons:

- Integrated WiFi card: although I had a few USB WiFi cards lying around, I liked the fact that the latest Raspberry Pi board not only included a WiFi card, but it also was supposed to offer much better performance than prior attempts.

- Better overall community support: the Raspberry Pi is just a much more popular

platform, so I felt more confident that down the road it would continue to get updates regardless of the media-center distribution I decided to use.

- Better for a DIY article: I worked on this project with an eye for writing about it. By selecting the Raspberry Pi, I chose hardware that you are more likely to have or at least will have an easier time getting.

- I'd likely need a USB IR port anyway: although the integrated IR port in the Odroid C2 was compelling, I wanted the ability to hide away the computer if that worked best when installing it. That meant I might not be able to mount it directly to the back of the TV and might have to run an IR receiver out to the TV anyway.

Along with a new Raspberry Pi 3 B+, I ordered a couple SD cards and chose a generic "MCE remote" that came with a USB IR receiver. MCE remotes are designed for Windows media PCs with a standard button arrangement for media servers, and they are generally well supported in Kodi distributions, so all of the buttons work out of the box.

Another key piece of hardware was the USB disk enclosure to store all of my files. I needed a lot of media storage in a relatively small package, so I opted for an 8TB Western Digital MyBook. You can find them in local electronic stores often for much cheaper than the same 8TB drive outside an enclosure. They also are powered by a 12V barrel connector, which makes it easy to swap out its AC adapter for a direct DC power supply that connects to one of the 12V DC adapters in the RV. Of course, if you can get by with less storage, you should consider a 2.5″ laptop hard drive instead, as they typically have much lower power requirements (and take up less space) compared to a 3.5″ drive.

Step 2: Media Server Software

I knew that I wanted to use some kind of pre-built Kodi distribution for my media center because I already was used to that interface at home. At home, I use OSMC, but for the RV system, I was choosing between it, OpenELEC and LibreELEC. OSMC is a Raspbian-based distribution that bundles in and configures Kodi for you. OpenELEC

and its more cutting-edge fork LibreELEC are more focused on optimization, and instead of being based on Raspbian, they have a more heavily customized and lightweight distribution. The end result is that OpenELEC and LibreELEC end up booting much faster.

At home, I typically leave my media server on all the time, even if the TV isn't on, so the amount of time OSMC takes to boot isn't an issue. In the RV, I wanted to save power, so I planned to power the Raspberry Pi off of the TV, so that it automatically turned on only when the TV was turned on. This made a fast boot time more compelling, and since LibreELEC touted cutting-edge support for my cutting-edge Raspberry Pi, I decided to try that first.

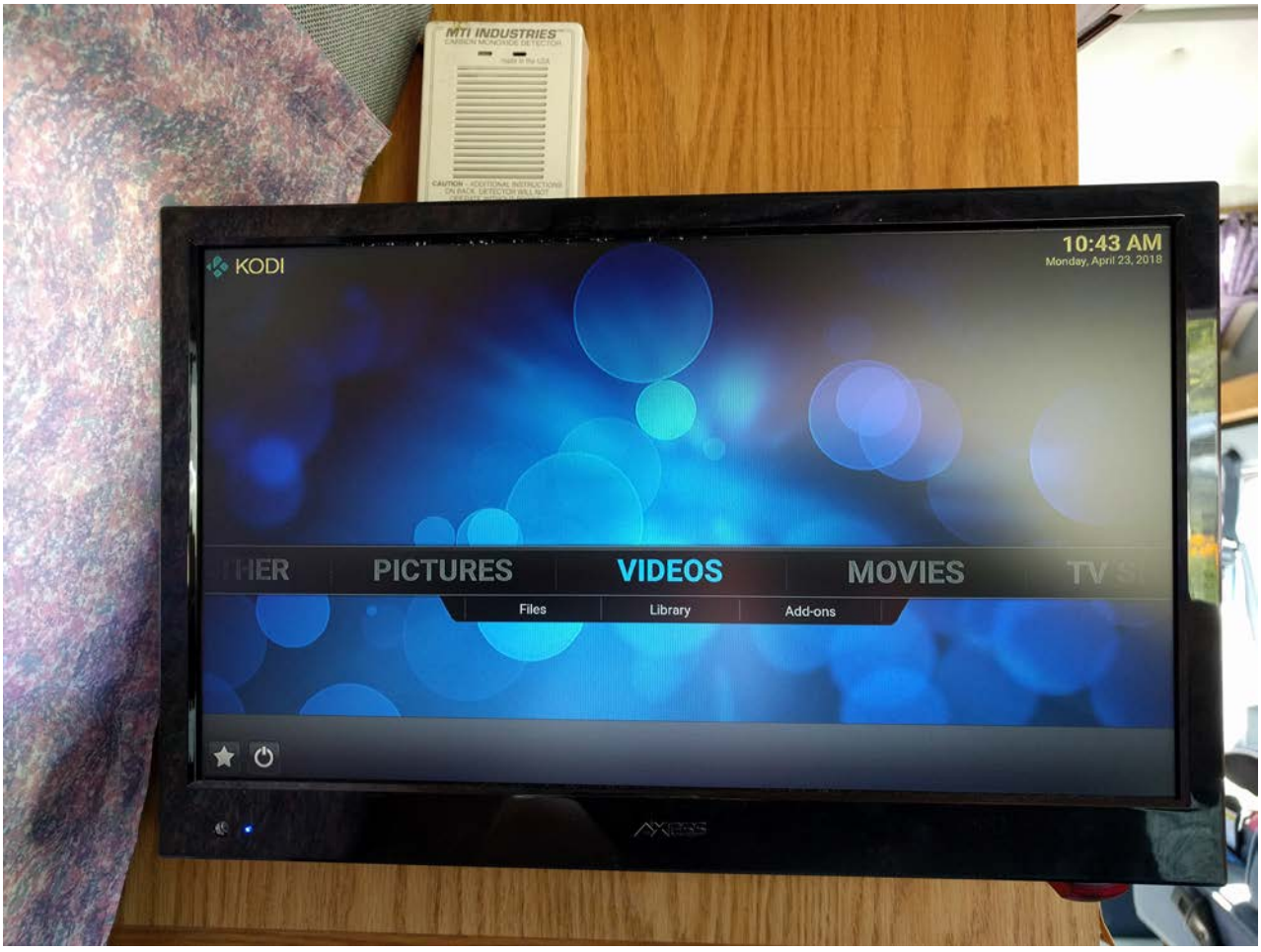


Figure 2. OSMC Running on My RV TV

The boot times for LibreELEC were great, but unfortunately, that stripped-down fast boot time also meant there weren't a whole lot of other tools available behind the scenes. Although LibreELEC does have the option to `ssh` in to the server, very few other packages are available. This meant I couldn't easily install rsync, which was critical to this project.

Ultimately, I decided on OSMC. Although the boot times are longer than LibreELEC, the fact that it is based on Raspbian means I can treat the underlying server like any other Debian system. This made customizing it to suit my backup and encryption needs much simpler. It also means I could add extra capabilities to this system in the future without springing for a separate Raspberry Pi.

Step 3: Media Server Configuration

Configuring the media server itself was pretty straightforward. I just copied the latest OSMC image from https://osmc.tv onto a microSD card per the instructions and booted from it just like with any other Raspbian install. On first boot, it resizes its partition to fill up the remaining space, and on the second boot, I went to the system settings page to make sure the Ethernet network was disabled and the WiFi network was connected to my access point. This is one time when I regretted that I have a long WiFi passphrase, because I had to enter it one character at a time via the arrow keys on a remote and the on-screen keyboard. (Note to self: next time bring a USB keyboard to the RV.)

Next, I `ssh`'d to this system over the wireless network from another computer with the default user name:password of osmc:osmc. This means my first step after I logged in was to change that password:

```
$ passwd
```

The next step was to configure the external 8TB hard drive. In most cases, this would just mean plugging it in, but given I planned to back up potentially sensitive documents to it, I wanted to add a layer of disk encryption, so if burglars came and stole my hard drive, they wouldn't be able to retrieve any data. First, I plugged in my

hard drive and checked `dmesg` output to confirm what device it appeared as. Because it's the only SATA-style drive connected, it should come up as /dev/sda, but you still want to be sure:

```
$ sudo dmesg | grep sd
. . .
[    4.805850] sd 0:0:0:0: [sda] Spinning up disk...
[    4.815562] sd 0:0:0:0: Attached scsi generic sg0 type 0
[   21.442041] sd 0:0:0:0: [sda] Very big device. Trying to
 ↪use READ CAPACITY(16).
[   21.442279] sd 0:0:0:0: [sda] 15628052480 512-byte logical
 ↪blocks: (8.00 TB/7.28 TiB)
[   21.442284] sd 0:0:0:0: [sda] 4096-byte physical blocks
[   21.442599] sd 0:0:0:0: [sda] Write Protect is off
[   21.442604] sd 0:0:0:0: [sda] Mode Sense: 47 00 10 08
[   21.442907] sd 0:0:0:0: [sda] No Caching mode page found
[   21.442911] sd 0:0:0:0: [sda] Assuming drive cache: write
 ↪through
[   21.443478] sd 0:0:0:0: [sda] Very big device. Trying to
 ↪use READ CAPACITY(16).
[   22.786525]  sda: sda1
[   22.787302] sd 0:0:0:0: [sda] Very big device. Trying to
 ↪use READ CAPACITY(16).
[   22.788287] sd 0:0:0:0: [sda] Attached SCSI disk
```

Now that I know it's showing up as /dev/sda, the next step is to set up encryption. To do this, I installed the cryptsetup package on my media server and then set up LUKS encryption on the external hard drive's only partition, /dev/sda1, that takes up all the space on the drive:

```
$ sudo apt install cryptsetup
$ sudo cryptsetup --verbose --verify-passphrase
 ↪luksFormat /dev/sda1
```

This process prompts you to set a decryption passphrase, so be sure to remember it and optionally write it down or store it in your password manager. Next, I need to open the device so I can assign it a name and format it:

```
$ sudo cryptsetup luksOpen /dev/sda1 crypt-sda1
$ sudo mkfs -t ext4 /dev/mapper/crypt-sda1
```

Now the drive is ready to be mounted, but the next time the system reboots it won't mount automatically, so now I need to configure that. Because I want the system to be able to mount this partition automatically, this means creating a key file readable only by root on this system, filling it with random contents and making that file an additional key to unlock the encrypted volume. Later I can point the system to this keyfile, but first let's set it up:

```
$ sudo dd if=/dev/urandom of=/root/keyfile bs=1024 count=4
$ sudo chmod 0400 /root/keyfile
$ sudo cryptsetup luksAddKey /dev/sda1 /root/keyfile
```

To configure the system to set up this device automatically at each boot, next I need to figure out what UUID the /dev/sda1 partition was assigned. Although I could use /dev/sda1, what if some day I add a second USB hard drive and it grabs that device first? I use the `blkid` command to get the UUID for the device:

```
sudo blkid /dev/sda1
/dev/sda1: UUID="074051d8-e239-408b-a3ba-ee28301bdee2"
 ↪TYPE="crypto_LUKS" PARTLABEL="My Book"
 ↪PARTUUID="a97a96bf-41a5-4358-9c33-3458ab36ddf4"
```

Now I've got the information I need to create a file called /etc/crypttab. This will give the system the information it requires to set up the LUKS device automatically each time it boots. My file has the following contents:

```
$ cat /etc/crypttab
```

```
# <target name> <source device>      <key file>    <options>
crypt-sda1 /dev/disk/by-uuid/074051d8-e239-408b-a3ba-ee28301bdee2
 ↪/root/keyfile luks
```

The first field is the name you want to assign to the device (this is what shows up in /dev/mapper). The next field is the full path to the raw device. As you can see, I pointed to the device by its UUID, which means using the path in /dev/disk/by-uuid/. The next field lets me point to the key file that I created, and the last field lets cryptsetup know that this disk is using LUKS for encryption.

OSMC now will mount this USB device at boot automatically without my having to add anything to /etc/fstab. By default, it mounts it in /media/UUID, but as that would be a pain to type for the rest of my scripts, I create a symlink under /mnt that's easier to type:

```
$ sudo ln -s /media/f6b0e02c-c08e-45f5-bb28-5a7c360d6f72/
 ↪/mnt/storage
```

Now I can use /mnt/storage whenever I want to access this device from my scripts.

Step 4: File Syncing

The next step was to set up my file-syncing script. The initial sync would take forever on WiFi, so I unplugged the drive and connected it directly to my NAS server, used the `luksOpen` command from above to open it with my passphrase, and then mounted it on a temporary directory (/mnt/temp in this example). Then I could create the directories for my backups and start the initial `rsync` commands:

```
$ sudo mkdir /mnt/temp/audio
$ sudo mkdir /mnt/temp/documents
$ sudo mkdir /mnt/temp/video
$ sudo rsync -avxH /mnt/storage/audio /mnt/temp/audio
$ sudo rsync -avxH /mnt/storage/documents /mnt/temp/documents
$ sudo rsync -avxH /mnt/storage/video /mnt/temp/video
```

This initial `rsync` took quite some time to complete, so in the mean time, I set up the rest of the syncing configuration. First, I added an entry to /etc/hosts that points to my RV for when it's on the network so I don't have to refer to it by IP:

```
192.168.1.50   rv
```

Because I want to `rsync` files as the root user on my NAS to the root user on my RV's media server, I end up breaking a general rule of mine and allow SSH logins as root. I make up for that though by setting up SSH keys for the default "osmc" user so I can disable password login altogether. From the NAS system I'll use to `ssh` in, I type the following as my regular user:

```
$ ssh-copy-id osmc@rv
```

This command will prompt me for my osmc user password one last time, and from that point on, I should be able to `ssh` back in to the RV without a password prompt. Now that I'm on the RV computer, it's a good time to create an SSH directory for the root user:

```
$ sudo mkdir -p /root/.ssh
$ sudo chmod 0700 /root/.ssh
$ sudo touch /root/.ssh/authorized_keys
$ sudo chmod 0600 /root/.ssh/authorized_keys
```

Since my `rsync` script will run as root (so it can make sure to have access to all the local files it is backing up), I also need to copy my root user's public SSH keys over. If your root user doesn't have SSH keys yet, just type the following to generate them:

```
$ sudo ssh-keygen -t rsa
```

Then copy and paste the contents of /root/.ssh/id_rsa.pub on your home file server to the /root/.ssh/authorized_keys file on your RV computer. Since I haven't enabled the root user yet (and the root user is disabled and doesn't have a password by default in OSMC), I have to use this method instead of `ssh-copy-id`.

Next I edit the /etc/ssh/sshd_config file on the RV server and make sure that the `PasswordAuthentication` option is commented out and the option `PermitRootLogin` is set to yes. Then I restart the SSH dæmon with:

```
$ sudo systemctl restart ssh
```

Note that if for some reason you didn't confirm that your user SSH keys were set up properly before this step and you end up being locked out, just power off the Raspberry Pi, remove the microSD card and insert it into another computer and undo your changes to the /etc/ssh/sshd_config file.

With the SSH dæmon restarted, I now should be able to become the root user on my home file server and `ssh` directly to root@rv without a password prompt.

The final step is to set up a script that synchronizes all of my files over to the RV server and add a cron job that tries to keep these files in sync every three hours. Because it's possible that it may take many hours sometimes to synchronize files between the systems over the wireless network, it's important to use a lock file so my script takes advantage of the `flock` command for this. Here is my /usr/local/bin/rvsync script that's on my home file server:

```
#!/bin/bash

date >>/tmp/rvsync-output
flock -n /tmp/rvsync.lock rsync -avxH /mnt/storage/audio/
 ↪rv:/mnt/storage/audio/ 2>/dev/null 1>>/tmp/rvsync-output
flock -n /tmp/rvsync.lock rsync -avxH /mnt/storage/documents/
 ↪rv:/mnt/storage/documents/ 2>/dev/null 1>>/tmp/rvsync-output
flock -n /tmp/rvsync.lock rsync -avxH /mnt/storage/video/
 ↪rv:/mnt/storage/video/ 2>/dev/null 1>>/tmp/rvsync-output

exit 0
```

The `flock` command sets up a lock file at /tmp/rvsync.lock. If another iteration of this script is running, the command doesn't run; otherwise, the `rsync` command runs. I add the current date and all `rsync` output to a /tmp/rsync-output file, so I can keep track of what was synced over in case I need to troubleshoot things or keep track of an rsync job that's in progress.

The `rsync` veterans among you might notice that I didn't include a `--delete` option in my `rsync` commands to remove files on the RV that were deleted at home. I did this mainly because I'm also using this as a kind of offsite backup, and I didn't want to risk a wayward delete command on my home NAS resulting in my backup being blown away as well. If storage starts to become a concern, I plan to run manual `rsync` commands with the `--delete` option added from time to time.

Finally, I created a file called /etc/cron.d/rvsync on my file server with the following contents:

```
0 */3 * * * root /usr/local/bin/rvsync
```

At the top of the hour every three hours, that script will attempt to run. If the RV isn't available, the script will fail silently; otherwise, it will sync over new files.

Once the initial sync progress was done, I unmounted and disconnected the USB drive and connected it back to my Raspberry Pi in the RV. Then I did a test run of the `rvsync` script. Although it worked, it was pretty slow due to the fact that the Raspberry Pi was pretty far away from the access point inside the house. Obviously, your circumstances might be different, but in my case, I decided to improve my reception by adding a WiFi extender.

WiFi extenders basically act as a kind of bridge between an existing access point and your computer. After some research, I selected the VONETS VAP11G-300, because it was small, relatively cheap and could be powered off USB. All I had to do to set this up was follow the simple instructions that came with the device. This involved plugging the power on, connecting to its default wireless network and then logging in to its web interface with its default credentials. At that point,

I used its wizard to configure it as a bridge, which involved pointing it at my existing access point and giving it a new SSID to use. Finally, I went back to my RV media server and updated its wireless configuration to point at this new access point. With the WiFi extender in place, I was able to increase my file transfer rates between three and four times what they were before, so I consider it a worthwhile investment. What's more, I always have the option to reconfigure this access point on the road to boost the signal to an RV park's WiFi.

Step 5: Installation

Now that all the hardware and software was set up, the final step was the physical installation. My RV is old enough that it came with a cabinet designed for a CRT TV. Since modern LCDs don't need all that space, I decided to reuse that cabinet as a pantry and mount an RV LCD TV to the wall beside that cabinet. I didn't want the screen to stick out and be visible from the front of the RV when it was put away, so I measured the cabinet and decided a 15″ screen would fit just about perfectly. Instead of going with a standard LCD designed for a computer, I opted for a TV designed for RVs that could be powered off a 12V DC car outlet and included a TV tuner. Some RV parks provide cable TV hookups, so I figured it might be fun to have that as an option. The direct DC power option was important too, because you ultimately waste power when you convert from AC to DC power (and even more so when you use an inverter to convert the RV's 12V DC to 110V AC only to convert it back to 12V DC), so I've made sure this entire system, including the external hard drive, could be powered off 12V DC. In the case of the hard drive, that meant scrounging around in a parts drawer for a universal 12V DC car adapter that happened to have the appropriate plug.

Another important consideration for my TV was making sure it not only had an HDMI input, but that it also had a USB port. I wanted to mount the Raspberry Pi to the back of the TV, and although those TV USB ports don't provide as much current as a proper USB power adapter, so far, I haven't had any issues. I ultimately settled on the AXESS TV1705-15 TV. I then used some of that blue tack material you might use to mount posters on a wall to mount my Raspberry Pi case to the back of the TV. I also used some of that tack material to mount the IR receiver on

the underside of the TV. To help with cable management, I bought a short one-foot HDMI cable. You can see the end result in Figure 3.

You might notice that I didn't mount the huge external USB drive to the back of the TV. Because I had to route power from inside the old TV cabinet anyway, I decided to keep the USB drive inside that cabinet and just run the USB cable along with the cable TV and TV power cables.

The final important decision I needed to make was which mounting bracket to use. I had seen pictures and videos of other people's RV TV mounts, and my main concern with most of them was because they used a cheaper mounting bracket, they ended up using some kind of bungee cord or other solution to strap the TV in so it wouldn't flop around while they were driving. I opted to spend a bit more on my wall-mount bracket to get one that had lockable segments. This way I could loosen a few thumb screws and position the TV where I wanted and then lock it back in place and it would stay that way. I also could lock it in place when the TV was put away (Figure 4) and not worry about it banging against cabinets when I went over bumps.

Conclusion

I've taken this set up on a couple trips so far, and I have to admit, it's been pretty great to have all of my media with me at all times—my son especially approves of having all his cartoons at hand. In conclusion, here are a few things to keep in mind if you set up a similar system.

1. Old RV converters provide "dirty" power.

Once thing I noticed with this system was that it worked great whenever I was running directly off battery, but when I plugged in to "shore power" (connecting to 110V power from an extension cord at my house), the TV wouldn't always power on. It turns out that the electronics in older RVs that convert 110V power back to 12V don't necessarily provide clean 12V power. Although dumb appliances don't mind, modern electronics sometimes do. The solution is to

Figure 3. The Back of the TV

upgrade your converter to a modern one or to use the system only when on battery power.

2. Do you leave the system on 24x7 when at home?

The big question you'll need to answer is whether you want to leave the RV media server on all the time when it's at home. If you are someone who leaves the RV plugged in when at home, that might work out great. Alternatively, if you have solar panels on top of your RV, you might be able to get away with leaving the system on all the time knowing that the solar panels are replenishing the power you draw from the house batteries. If you're relying on the house batteries by

Figure 4. The Mounting Bracket Collapsed and Ready for Travel

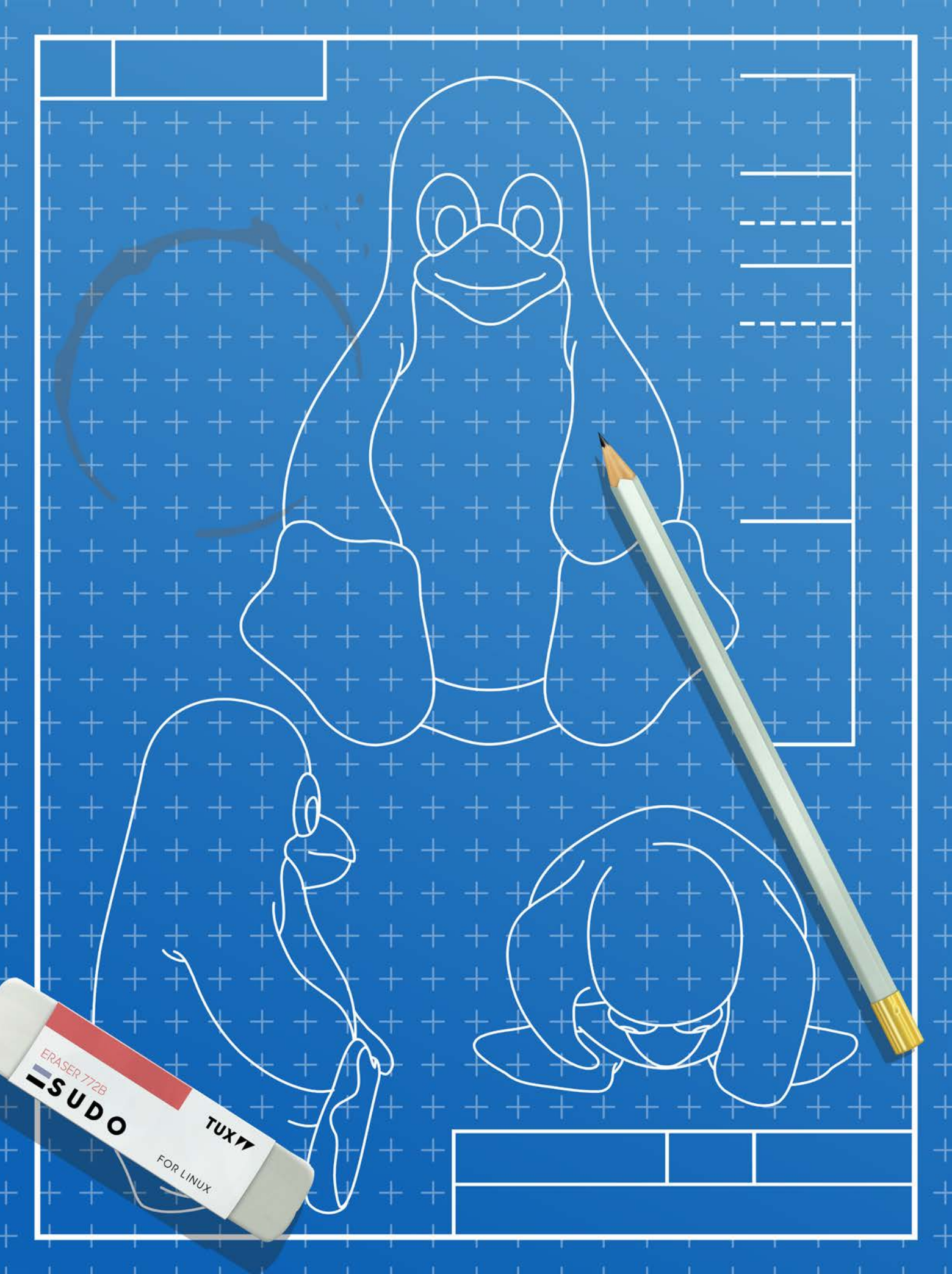
themselves to power your server, you'll probably drain your battery after a day or two.

Because I don't leave my RV powered on all the time, it means I have to take a more active approach to syncing my files. Every week or two, I turn on my RV media server for a few hours and let the cron job kick in and sync files over. When I know I'm going to be going on a trip, I tend to power the RV ahead of time to give the fridge a chance to cool down, so I also take that opportunity to sync files over. For now, this is a fair compromise for me until I upgrade my converter, and then I might consider leaving the server on all the time. If you are going to rely on this as your main offsite backup, that may not be frequent

enough for you.

I've been pleased with this set up so far. It's nice to watch movies when I'm camped far enough away from civilization that streaming isn't an option. Even more than that, it's nice to have an additional backup of my important files that are near enough allowing me to access them if I need to, but on a mobile platform, so in the event of a fire or some other disaster, I could leave at a moment's notice and have everything I need with me. ■

Kyle Rankin is a Tech Editor and columnist at *Linux Journal* and the Chief Security Officer at Purism. He is the author of *Linux Hardening in Hostile Networks, DevOps Troubleshooting, The Official Ubuntu Server Book, Knoppix Hacks, Knoppix Pocket Reference, Linux Multimedia Hacks* and *Ubuntu Hacks*, and also a contributor to a number of other O'Reilly books. Rankin speaks frequently on security and open-source software including at BsidesLV, O'Reilly Security Conference, OSCON, SCALE, CactusCon, Linux World Expo and Penguicon. You can follow him at @kylerankin.


ERASER 772B
SUDO
TUX
FOR LINUX

DIY: Build a Custom Minimal Linux Distribution from Source

Follow along with this step-by-step guide to build your own distribution from source and learn how it installs, loads and runs.

By Petros Koutoupis

When working with Linux, you easily can download any of the most common distributions to install and configure—be it Ubuntu, Debian, Fedora, OpenSUSE or something entirely different. And although you should give several distributions a spin, building your own custom, minimal Linux distribution is also a beneficial and wonderful learning exercise.

When I say "build a custom and minimal Linux distribution", I mean from *source* packages—that is, start with a cross-compiling toolchain and then build a target image to install on a physical or virtual hard disk drive (HDD).

So, when I think of the ultimate Do-It-Yourself (DIY) guide related to Linux, it's got to be exactly this: *building a Linux distribution from source*. The entire process will take at least a couple hours on a decently powered host machine.

If you follow along with this exercise, you'll learn what it takes to build a custom

distribution, and you'll also learn how that distribution installs, loads and runs. You can run this exercise on either a physical or virtual machine.

I'd be lying if I said that this process wasn't partly inspired by the wonderful Linux From Scratch (LSF) project. The LSF project proved to be an essential tool in my understanding of how a standard Linux operating system is built and functions. Using a similar philosophy, I hope to instill some of the same wisdom to you, the reader, if you'd like to follow along.

Terms

- **Host:** the *host* signifies the very machine on which you'll be doing the vast majority of the work, including cross compilation and installation of the target image.

- **Target:** the *target* is the final cross-compiled operating system that you'll be building from source packages. It'll be built using the cross compiler on the host machine.

- **Cross-compiler:** you'll be building and using a *cross compiler* to create the *target* image on the *host* machine. A cross compiler is built to run on a host machine, but it's used to compile for a target architecture or microprocessor that isn't compatible with the host machine.

Prerequisites and Tools

To continue with this tutorial, you'll need to have GCC, make, ncurses, Perl and grub tools (specifically grub-install) installed on the host machine.

In order to build anything, you'll also need to download and build all the packages for the cross compiler and the target image. I'm using the following open-source packages and versions for this tutorial:

- binutils-2.30.tar.xz
- busybox-1.28.3.tar.bz2
- clfs-embedded-bootscripts-1.0-pre5.tar.bz2
- gcc-7.3.0.tar.xz
- glibc-2.27.tar.xz
- gmp-6.1.2.tar.bz2
- linux-4.16.3.tar.xz
- mpc-1.1.0.tar.gz
- mpfr-4.0.1.tar.xz
- zlib-1.2.11.tar.gz

Configuring the Environment

Before beginning this process, you need to configure the build environment. First, turn on Bash hash functions:

```
$ set +h
```

Make sure that newly created files/directories are writable only by the owner (for example, the currently logged in user account):

```
$ umask 022
```

You'll use your home directory as the main build directory. (this isn't a requirement). This is where the cross-compilation toolchain and target image will be installed and put into the lj-os subdirectory. If you prefer to install it elsewhere, make the adjustment to the code section below:

```
$ export LJOS=~/lj-os
$ mkdir -pv ${LJOS}
```

Finally, export some remaining variables:

```
$ export LC_ALL=POSIX
$ export PATH=${LJOS}/cross-tools/bin:/bin:/usr/bin
```

After setting the above environment variables, create the target image's filesystem hierarchy:

```
$ mkdir -pv ${LJOS}/{bin,boot{,grub},dev,{etc/,}opt,home,
↪lib/{firmware,modules},lib64,mnt}
$ mkdir -pv ${LJOS}/{proc,media/{floppy,cdrom},sbin,srv,sys}
$ mkdir -pv ${LJOS}/var/{lock,log,mail,run,spool}
$ mkdir -pv ${LJOS}/var/{opt,cache,lib/{misc,locate},local}
$ install -dv -m 0750 ${LJOS}/root
$ install -dv -m 1777 ${LJOS}{/var,}/tmp
$ install -dv ${LJOS}/etc/init.d
$ mkdir -pv ${LJOS}/usr/{,local/}{bin,include,lib{,64},sbin,src}
$ mkdir -pv ${LJOS}/usr/{,local/}share/{doc,info,locale,man}
$ mkdir -pv ${LJOS}/usr/{,local/}share/{misc,terminfo,zoneinfo}
$ mkdir -pv ${LJOS}/usr/{,local/}share/man/man{1,2,3,4,5,6,7,8}
```

```
$ for dir in ${LJOS}/usr{,/local}; do
     ln -sv share/{man,doc,info} ${dir}
   done
```

This directory tree is based on the Filesystem Hierarchy Standard (FHS), which is defined and hosted by the Linux Foundation:

Create the directory for a cross-compilation toolchain:

```
$ install -dv ${LJOS}/cross-tools{,/bin}
```

Use a symlink to /proc/mounts to maintain a list of mounted filesystems properly in the /etc/mtab file:

```
$ ln -svf ../proc/mounts ${LJOS}/etc/mtab
```

Then create the /etc/passwd file, listing the root user account (note: for now, you won't be setting the account password; you'll do that after booting up into the target image for the first time):

```
$ cat > ${LJOS}/etc/passwd << "EOF"
root::0:0:root:/root:/bin/ash
EOF
```

Create the /etc/group file with the following command:

```
$ cat > ${LJOS}/etc/group << "EOF"
root:x:0:
bin:x:1:
sys:x:2:
kmem:x:3:
tty:x:4:
daemon:x:6:
```

```
disk:x:8:
dialout:x:10:
video:x:12:
utmp:x:13:
usb:x:14:
EOF
```

The target system's /etc/fstab:

```
$ cat > ${LJOS}/etc/fstab << "EOF"
# file system  mount-point  type   options          dump  fsck
#                                                        order

rootfs          /              auto    defaults        1      1
proc            /proc          proc    defaults        0      0
sysfs           /sys           sysfs   defaults        0      0
devpts          /dev/pts       devpts  gid=4,mode=620  0      0
tmpfs           /dev/shm       tmpfs   defaults        0      0
EOF
```

The target system's /etc/profile to be used by the Almquist shell (ash) once the user is logged in to the target machine:

```
$ cat > ${LJOS}/etc/profile << "EOF"
export PATH=/bin:/usr/bin

if [ 'id -u' -eq 0 ] ; then
        PATH=/bin:/sbin:/usr/bin:/usr/sbin
        unset HISTFILE
fi


# Set up some environment variables.
```

```
export USER='id -un'
export LOGNAME=$USER
export HOSTNAME='/bin/hostname'
export HISTSIZE=1000
export HISTFILESIZE=1000
export PAGER='/bin/more '
export EDITOR='/bin/vi'
EOF
```

The target machine's hostname (you can change this any time):

```
$ echo "ljos-test" > ${LJOS}/etc/HOSTNAME
```

And, /etc/issue, which will be displayed prominently at the login prompt:

```
$ cat > ${LJOS}/etc/issue<< "EOF"
Linux Journal OS 0.1a
Kernel \r on an \m

EOF
```

You won't use systemd here (this wasn't a political decision; it's due to convenience and for simplicity's sake). Instead, you'll use the basic `init` process provided by BusyBox. This requires that you define an /etc/inittab file:

```
$ cat > ${LJOS}/etc/inittab<< "EOF"
::sysinit:/etc/rc.d/startup

tty1::respawn:/sbin/getty 38400 tty1
tty2::respawn:/sbin/getty 38400 tty2
tty3::respawn:/sbin/getty 38400 tty3
tty4::respawn:/sbin/getty 38400 tty4
tty5::respawn:/sbin/getty 38400 tty5
```

```
tty6::respawn:/sbin/getty 38400 tty6

::shutdown:/etc/rc.d/shutdown
::ctrlaltdel:/sbin/reboot
EOF
```

Also as a result of leveraging BusyBox to simplify some of the most common Linux system functionality, you'll use `mdev` instead of `udev`, which requires you to define the following /etc/mdev.conf file:

```
$ cat > ${LJOS}/etc/mdev.conf<< "EOF"
# Devices:
# Syntax: %s %d:%d %s
# devices user:group mode

# null does already exist; therefore ownership has to
# be changed with command
null    root:root 0666  @chmod 666 $MDEV
zero    root:root 0666
grsec   root:root 0660
full    root:root 0666

random  root:root 0666
urandom root:root 0444
hwrandom root:root 0660

# console does already exist; therefore ownership has to
# be changed with command
console root:tty 0600 @mkdir -pm 755 fd && cd fd && for x
 ↪in 0 1 2 3 ; do ln -sf /proc/self/fd/$x $x; done

kmem    root:root 0640
mem     root:root 0640
```

```
port    root:root 0640
ptmx    root:tty 0666

# ram.*
ram([0-9]*)     root:disk 0660 >rd/%1
loop([0-9]+)    root:disk 0660 >loop/%1
sd[a-z].*       root:disk 0660 */lib/mdev/usbdisk_link
hd[a-z][0-9]*   root:disk 0660 */lib/mdev/ide_links

tty             root:tty 0666
tty[0-9]        root:root 0600
tty[0-9][0-9]   root:tty 0660
ttyO[0-9]*      root:tty 0660
pty.*           root:tty 0660
vcs[0-9]*       root:tty 0660
vcsa[0-9]*      root:tty 0660

ttyLTM[0-9]     root:dialout 0660 @ln -sf $MDEV modem
ttySHSF[0-9]    root:dialout 0660 @ln -sf $MDEV modem
slamr           root:dialout 0660 @ln -sf $MDEV slamr0
slusb           root:dialout 0660 @ln -sf $MDEV slusb0
fuse            root:root  0666

# misc stuff
agpgart         root:root 0660  >misc/
psaux           root:root 0660  >misc/
rtc             root:root 0664  >misc/

# input stuff
event[0-9]+     root:root 0640 =input/
ts[0-9]         root:root 0600 =input/

# v4l stuff
```

```
vbi[0-9]        root:video 0660 >v4l/
video[0-9]      root:video 0660 >v4l/

# load drivers for usb devices
usbdev[0-9].[0-9]       root:root 0660 */lib/mdev/usbdev
usbdev[0-9].[0-9]_.*    root:root 0660
EOF
```

You'll need to create a /boot/grub/grub.cfg for the GRUB bootloader that will be installed on the target machine's physical or virtual HDD (note: the kernel image defined in this file needs to reflect the image built and installed on the target machine):

```
$ cat > ${LJOS}/boot/grub/grub.cfg<< "EOF"
petros@ubuntu:/mnt/amd64$ vim ${LJOS}/boot/grub/grub.cfg

set default=0
set timeout=5

set root=(hd0,1)

menuentry "Linux Journal OS 0.1a" {
        linux   /boot/vmlinuz-4.16.3 root=/dev/sda1 ro quiet
}
EOF
```

Finally, initialize the log files and give them proper permissions:

```
$ touch ${LJOS}/var/run/utmp ${LJOS}/var/log/{btmp,lastlog,wtmp}
$ chmod -v 664 ${LJOS}/var/run/utmp ${LJOS}/var/log/lastlog
```

Building the Cross Compiler

If you recall, the cross compiler is a toolchain of various compilation tools

built for the system on which it's executing but designed to compile for an architecture or microprocessor that's not necessarily compatible with the system on which you're using it. In my environment, I'm running on a 64-bit x86 architecture (x86-64) and will be cross compiling to a generic x86-64 target architecture. Sure, this section is somewhat redundant considering the environment I am running in, but the tutorial is designed to ensure that you are able to build for an x86-64 target, regardless of the machine type that you are using (for example, PowerPC, ARM, x86 and so on).

You never can be too sure with what is set in a currently running environment, which is why you'll unset the following C and C++ flags:

```
$ unset CFLAGS
$ unset CXXFLAGS
```

Next, define the most vital parts of the host/target variables needed to create the cross-compiler toolchain and target image:

```
$ export LJOS_HOST=$(echo ${MACHTYPE} | sed "s/-[^-]*/-cross/")
$ export LJOS_TARGET=x86_64-unknown-linux-gnu
$ export LJOS_CPU=k8
$ export LJOS_ARCH=$(echo ${LJOS_TARGET} | sed -e
 ↪'s/-.*//' -e 's/i.86/i386/')
$ export LJOS_ENDIAN=little
```

Kernel Headers

The kernel's standard header files need to be installed for the cross compiler. Uncompress the kernel tarball and change into its directory. Then run:

```
$ make mrproper
$ make ARCH=${LJOS_ARCH} headers_check && \
make ARCH=${LJOS_ARCH} INSTALL_HDR_PATH=dest headers_install
$ cp -rv dest/include/* ${LJOS}/usr/include
```

Binutils

Binutils contains a linker, assembler and other tools needed to handle compiled object files. Uncompress the tarball. Then create the binutils-build directory and change into it:

```
$ mkdir binutils-build
$ cd binutils-build/
```

Then run:

```
$ ../binutils-2.30/configure --prefix=${LJOS}/cross-tools \
--target=${LJOS_TARGET} --with-sysroot=${LJOS} \
--disable-nls --enable-shared --disable-multilib
$ make configure-host && make
$ ln -sv lib ${LJOS}/cross-tools/lib64
$ make install
```

Copy over the following header file to the target's filesystem:

```
$ cp -v ../binutils-2.30/include/libiberty.h ${LJOS}/usr/include
```

GCC (Static) Before building the final cross-compiler toolchain, you first must build a statically compiled toolchain to build the C library (glibc) to which the final GCC cross compiler will link.

Uncompress the GCC tarball, and then uncompress the following packages and move them into the GCC root directory:

```
$ tar xjf gmp-6.1.2.tar.bz2
$ mv gmp-6.1.2 gcc-7.3.0/gmp
$ tar xJf mpfr-4.0.1.tar.xz
$ mv mpfr-4.0.1 gcc-7.3.0/mpfr
$ tar xzf mpc-1.1.0.tar.gz
$ mv mpc-1.1.0 gcc-7.3.0/mpc
```

Now create a gcc-static directory and change into it:

```
$ mkdir gcc-static
$ cd gcc-static/
```

Run the following commands:

```
$ AR=ar LDFLAGS="-Wl,-rpath,${HYVEOS}/cross-tools/lib" \
../gcc-7.3.0/configure --prefix=${LJOS}/cross-tools \
--build=${LJOS_HOST} --host=${LJOS_HOST} \
--target=${LJOS_TARGET} \
--with-sysroot=${LJOS}/target --disable-nls \
--disable-shared \
--with-mpfr-include=$(pwd)/../gcc-7.3.0/mpfr/src \
--with-mpfr-lib=$(pwd)/mpfr/src/.libs \
--without-headers --with-newlib --disable-decimal-float \
--disable-libgomp --disable-libmudflap --disable-libssp \
--disable-threads --enable-languages=c,c++ \
--disable-multilib --with-arch=${LJOS_CPU}
$ make all-gcc all-target-libgcc && \
make install-gcc install-target-libgcc
$ ln -vs libgcc.a '${LJOS_TARGET}-gcc -print-libgcc-file-name |
 ↪sed 's/libgcc/&_eh/''
```

Do not delete these directories; you'll need to come back to them from the final version of GCC.

Glibc Uncompress the glibc tarball. Then create the glibc-build directory and change into it:

```
$ mkdir glibc-build
$ cd glibc-build/
```

Configure the following build flags:

```
$ echo "libc_cv_forced_unwind=yes" > config.cache
$ echo "libc_cv_c_cleanup=yes" >> config.cache
$ echo "libc_cv_ssp=no" >> config.cache
$ echo "libc_cv_ssp_strong=no" >> config.cache
```

Then run:

```
$ BUILD_CC="gcc" CC="${LJOS_TARGET}-gcc" \
AR="${LJOS_TARGET}-ar" \
RANLIB="${LJOS_TARGET}-ranlib" CFLAGS="-O2" \
../glibc-2.27/configure --prefix=/usr \
--host=${LJOS_TARGET} --build=${LJOS_HOST} \
--disable-profile --enable-add-ons --with-tls \
--enable-kernel=2.6.32 --with-__thread \
--with-binutils=${LJOS}/cross-tools/bin \
--with-headers=${LJOS}/usr/include \
--cache-file=config.cache
$ make && make install_root=${LJOS}/ install
```

GCC (Final) As I mentioned previously, you'll now build the final GCC cross compiler that will link to the C library built and installed in the previous step. Create the gcc-build directory and change into it:

```
$ mkdir gcc-build
$ cd gcc-build/
```

Then run:

```
$ AR=ar LDFLAGS="-Wl,-rpath,${LJOS}/cross-tools/lib" \
../gcc-7.3.0/configure --prefix=${LJOS}/cross-tools \
--build=${LJOS_HOST} --target=${LJOS_TARGET} \
```

```
--host=${LJOS_HOST} --with-sysroot=${LJOS} \
--disable-nls --enable-shared \
--enable-languages=c,c++ --enable-c99 \
--enable-long-long \
--with-mpfr-include=$(pwd)/../gcc-7.3.0/mpfr/src \
--with-mpfr-lib=$(pwd)/mpfr/src/.libs \
--disable-multilib --with-arch=${LJOS_CPU}
$ make && make install
$ cp -v ${LJOS}/cross-tools/${LJOS_TARGET}/lib64/
↪libgcc_s.so.1 ${LJOS}/lib64
```

Now that you've built the cross compiler, you need to adjust and export the following variables:

```
$ export CC="${LJOS_TARGET}-gcc"
$ export CXX="${LJOS_TARGET}-g++"
$ export CPP="${LJOS_TARGET}-gcc -E"
$ export AR="${LJOS_TARGET}-ar"
$ export AS="${LJOS_TARGET}-as"
$ export LD="${LJOS_TARGET}-ld"
$ export RANLIB="${LJOS_TARGET}-ranlib"
$ export READELF="${LJOS_TARGET}-readelf"
$ export STRIP="${LJOS_TARGET}-strip"
```

Building the Target Image

The hard part is now complete—you have the cross compiler. Now, let's focus on building the components that will be installed on the target image. This includes various libraries and utilities and, of course, the Linux kernel itself.

BusyBox BusyBox is one of my all-time favorite open-source projects. The project advertises itself to be the Swiss Army knife of open-source utilities, and that's probably the best description one could give the project. BusyBox combines a large collection of tiny versions of the most commonly used Linux

utilities into a single distributed package. Those tools range from common binaries, text editors and command-line shells to filesystem and networking utilities, process management tools and many more.

Uncompress the tarball and change into its directory. Then load the default compilation configuration template:

```
$ make CROSS_COMPILE="${LJOS_TARGET}-" defconfig
```

The default configuration template will enable the compilation of a default defined set of utilities and libraries. You can enable/disable whatever you see fit by running `menuconfig`:

```
$ make CROSS_COMPILE="${LJOS_TARGET}-" menuconfig
```

Compile and install the package:

```
$ make CROSS_COMPILE="${LJOS_TARGET}-"
$ make CROSS_COMPILE="${LJOS_TARGET}-" \
CONFIG_PREFIX="${LJOS}" install
```

Install the following Perl script, as you'll need it for the kernel build below:

```
$ cp -v examples/depmod.pl ${LJOS}/cross-tools/bin
$ chmod 755 ${LJOS}/cross-tools/bin/depmod.pl
```

The Linux Kernel Change into the kernel package directory and run the following to set the default x86-64 configuration template:

```
$ make ARCH=${LJOS_ARCH} \
CROSS_COMPILE=${LJOS_TARGET}- x86_64_defconfig
```

This will define a minimum set of modules and settings for the compilation process.

You most likely will need to make the proper adjustments for the target machine's environment. This includes enabling modules for storage and networking controllers and more. You can do that with the `menuconfig` option:

```
$ make ARCH=${LJOS_ARCH} \
CROSS_COMPILE=${LJOS_TARGET}- menuconfig
```

For instance, I'm going to be running this target image in a VirtualBox virtual machine where it will rely on an Intel `e1000` networking module (defaulted in defconfig) and an LSI `mpt2sas` storage controller for the operating system drive. For the sake of simplicity, these modules are configured to be compiled statically into the kernel image—that is, set to `*` instead of `m`. *Be sure to review what's needed and enable it, or your target environment will not operate properly when booted.*

Compile and install the kernel:

```
$ make ARCH=${LJOS_ARCH} \
CROSS_COMPILE=${LJOS_TARGET}-
$ make ARCH=${LJOS_ARCH} \
CROSS_COMPILE=${LJOS_TARGET}- \
INSTALL_MOD_PATH=${LJOS} modules_install
```

You'll need to copy a few files into the /boot directory for GRUB:

```
$ cp -v arch/x86/boot/bzImage ${LJOS}/boot/vmlinuz-4.16.3
$ cp -v System.map ${LJOS}/boot/System.map-4.16.3
$ cp -v .config ${LJOS}/boot/config-4.16.3
```

Then run the previously installed Perl script provided by the BusyBox package:

```
$ ${LJOS}/cross-tools/bin/depmod.pl \
-F ${LJOS}/boot/System.map-4.16.3 \
-b ${LJOS}/lib/modules/4.16.3
```

The Bootscripts The Cross Linux From Scratch (CLFS) project (a fork of the original LFS project) provides a wonderful set of bootscripts that I use here for simplicity's sake. Uncompress the package and change into its directory. Out of box, one of the package's makefiles contains a line that may not be compatible with your current working shell. Apply the following changes to the package's root Makefile to ensure that you don't experience any issues with package installation:

```
@@ -19,7 +19,9 @@ dist:
        rm -rf "dist/clfs-embedded-bootscripts-$(VERSION)"

 create-dirs:
-       install -d -m ${DIRMODE}
 ↪${EXTDIR}/rc.d/{init.d,start,stop}
+       install -d -m ${DIRMODE} ${EXTDIR}/rc.d/init.d
+       install -d -m ${DIRMODE} ${EXTDIR}/rc.d/start
+       install -d -m ${DIRMODE} ${EXTDIR}/rc.d/stop

 install-bootscripts: create-dirs
        install -m ${CONFMODE} clfs/rc.d/init.d/functions
         ↪${EXTDIR}/rc.d/init.d/
```

Then run the following commands to install and configure the target environment appropriately:

```
$ make DESTDIR=${LJOS}/ install-bootscripts
$ ln -sv ../rc.d/startup ${LJOS}/etc/init.d/rcS
```

Zlib Now you're at the very last package for this tutorial. Zlib isn't a requirement, but it serves as a great guide for other packages you may want to install for your environment. Feel free to skip this step if you'd rather format and configure the physical or virtual HDD.

Uncompress the Zlib tarball and change into its directory. Then configure, build and

install the package:

```
$ sed -i 's/-O3/-Os/g' configure
$ ./configure --prefix=/usr --shared
$ make && make DESTDIR=${LJOS}/ install
```

Now, because some packages may look for Zlib libraries in the /lib directory instead of the /lib64 directory, apply the following changes:

```
$ mv -v ${LJOS}/usr/lib/libz.so.* ${LJOS}/lib
$ ln -svf ../../lib/libz.so.1 ${LJOS}/usr/lib/libz.so
$ ln -svf ../../lib/libz.so.1 ${LJOS}/usr/lib/libz.so.1
$ ln -svf ../lib/libz.so.1 ${LJOS}/lib64/libz.so.1
```

Installing the Target Image

All of the cross compilation is complete. Now you have everything you need to install the entire cross-compiled operating system to either a physical or virtual drive, but before doing that, let's not tamper with the original target build directory by making a copy of it:

```
$ cp -rf ljos/ ljos-copy
```

Use this copy for the remainder of this tutorial. Remove some of the now unneeded directories:

```
$ rm -rfv ${LJOS}-copy/cross-tools
$ rm -rfv ${LJOS}-copy/usr/src/*
```

Followed by the now unneeded statically compiled library files (if any):

```
$ FILES="$(ls ${LJOS}-copy/usr/lib64/*.a)"
$ for file in $FILES; do
> rm -f $file
> done
```

Now strip all debug symbols from the installed binaries. This will reduce overall file sizes and keep the target image's overall footprint to a minimum:

```
$ find ${LJOS}-copy/{,usr/}{bin,lib,sbin} -type f
 ↪-exec sudo strip --strip-debug '{}' ';'
$ find ${LJOS}-copy/{,usr/}lib64 -type f -exec sudo
 ↪strip --strip-debug '{}' ';'
```

Finally, change file ownerships and create the following nodes:

```
$ sudo chown -R root:root ${LJOS}-copy
$ sudo chgrp 13 ${LJOS}-copy/var/run/utmp
 ↪${LJOS}-copy/var/log/lastlog
$ sudo mknod -m 0666 ${LJOS}-copy/dev/null c 1 3
$ sudo mknod -m 0600 ${LJOS}-copy/dev/console c 5 1
$ sudo chmod 4755 ${LJOS}-copy/bin/busybox
```

Change into the target copy directory to create a tarball of the entire operating system image:

```
$ cd ljos-copy/
$ sudo tar cfJ ../ljos-build-21April2018.tar.xz *
```

Notice how the target image is less than 60MB. You built that—a minimal Linux operating system that occupies less than 60MB of disk space:

```
$ sudo du -h|tail -n1
58M     .
```

And, that same operating system compresses to less than 20MB:

```
$ ls -lh ljos-build-21April2018.tar.xz
-rw-r--r-- 1 root root 18M Apr 21 15:31
 ↪ljos-build-21April2018.tar.xz
```

For the rest of this tutorial, you'll need a disk drive. It will need to enumerate as a traditional block device (in my case, it's /dev/sdd):

```
$ cat /proc/partitions |grep sdd
   8       48     256000 sdd
```

That block device will need to be partitioned. A single partition should suffice, and you can use any one of a number of partition utilities, including `fdisk` or `parted`. Once that partition is created and detected by the host system, format the partition with an ext4 filesystem, mount that partition to a staging area and change into that directory:

```
$ sudo mkfs.ext4 /dev/sdd1
$ sudo mkdir tmp
$ sudo mount /dev/sdd1 tmp/
$ cd tmp/
```

Uncompress the operating system tarball of the entire target operating system into the root of the staging directory:

```
$ sudo tar xJf ../ljos-build-21April2018.tar.xz
```

Now run `grub-install` to install all the necessary modules and boot records to the volume:

```
$ sudo grub-install --root-directory=/home/petros/tmp/ /dev/sdd
```

The `--root-directory` parameter defines the absolute path of the staging directory, while the last parameter is the block device without the partition's label.

Booting Up for the First Time

Now you're officially done. Install the HDD to the physical or virtual machine (as the primary disk drive) and power it up. You immediately will be greeted by the GRUB bootloader (Figure 1).

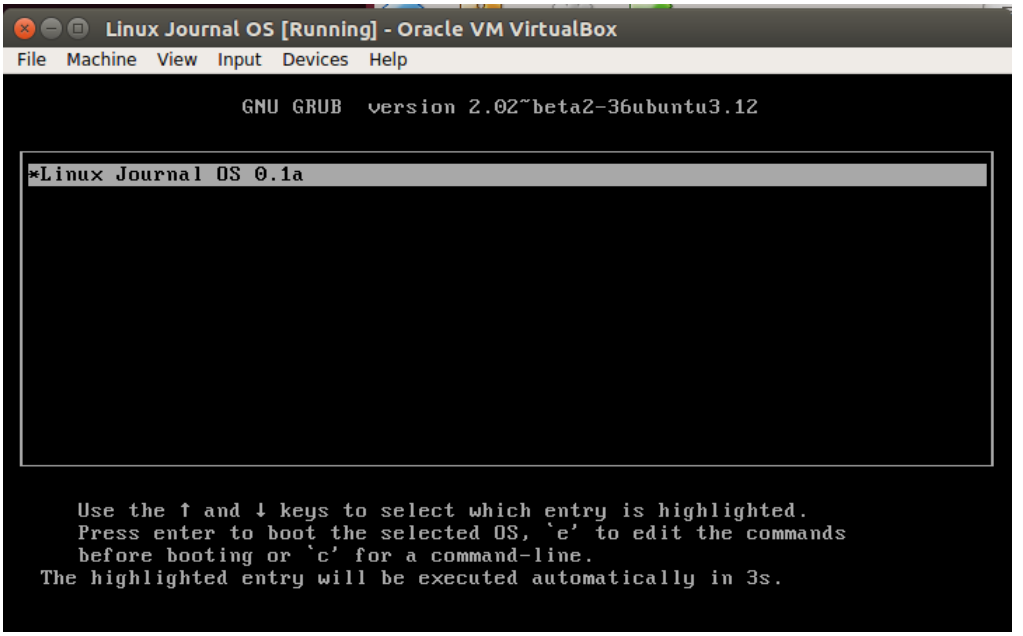


Figure 1. The GRUB Bootloader

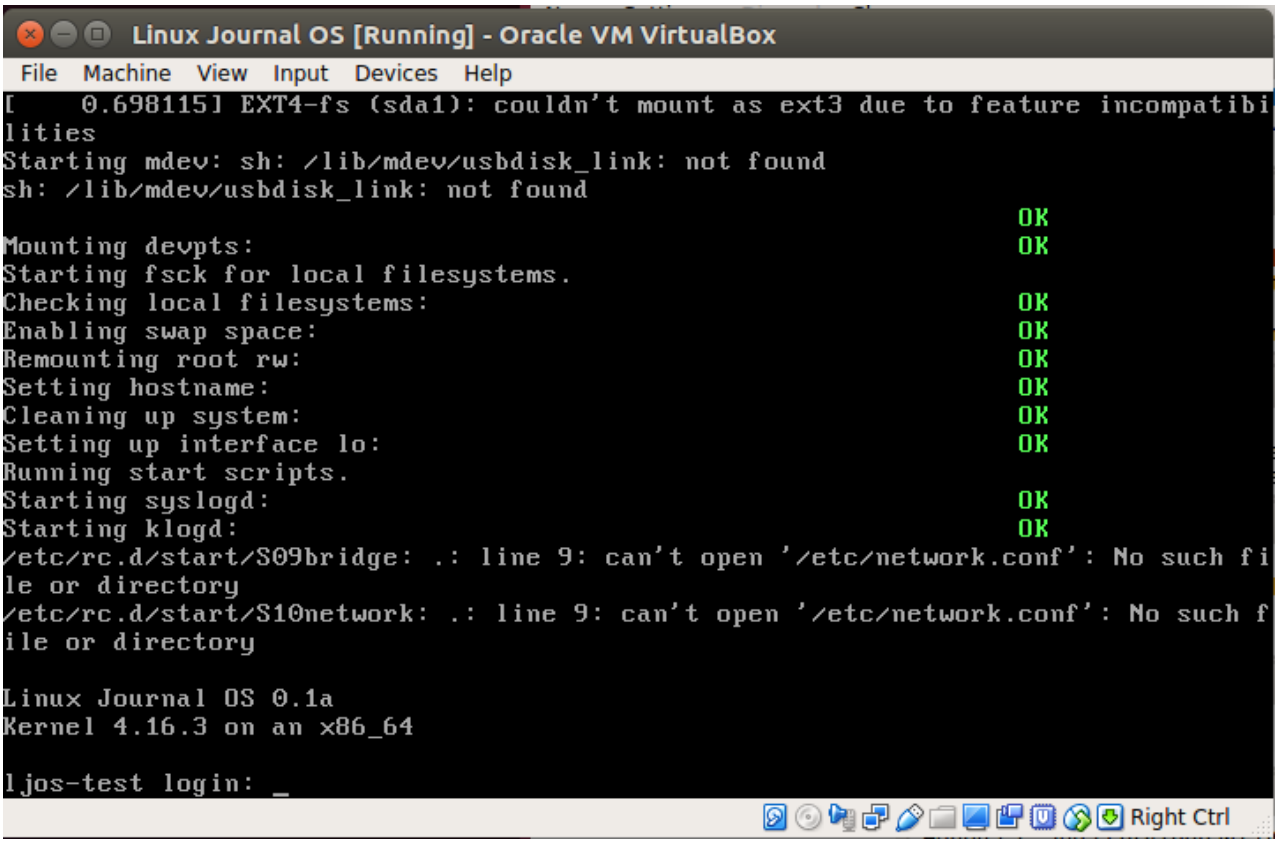


Figure 2. The User Login Prompt

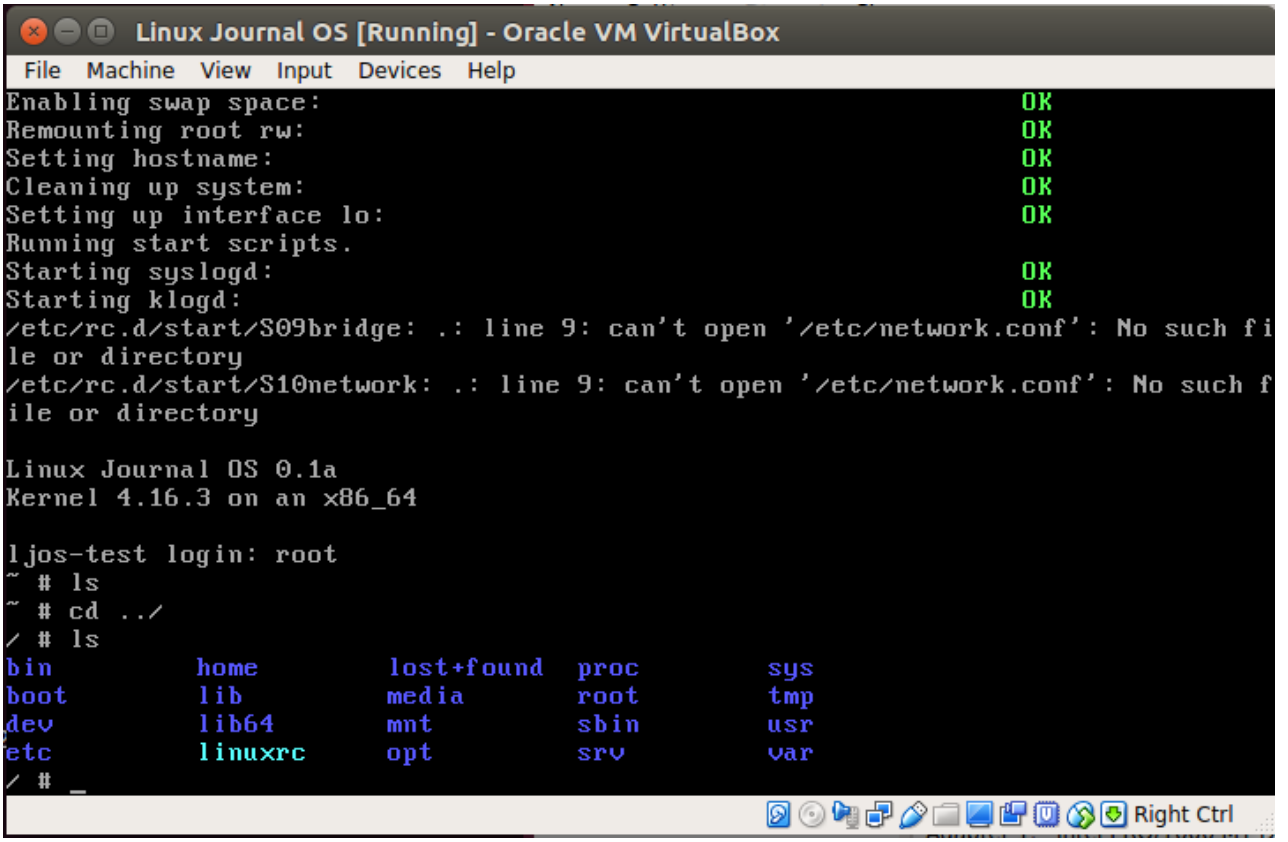


Figure 3. Executing a Few Simple Tasks

And within one second (yes, you read that correctly, one second), you'll be at the operating system's login prompt (Figure 2).

You'll notice a couple boot "error" and warning messages. This is because you're missing a couple files. You can correct that as you continue to learn the environment and build more packages into the operating system.

If you recall, you never set a root password. This was intentional. Log in as root, and you'll immediately fall into a shell without needing to input a password. You can change this behavior by using BusyBox's `passwd` command, which should have been built in to this image.

Enjoy!

Next Steps

So, where does this leave you now? You were able to build a custom Linux distribution for the generic x86-64 architecture from open-source packages and load into it successfully. Employing the same cross-compilation toolchain, you can use a similar process to build more utilities and libraries into the operating system, such as networking utilities, storage volume management frameworks and more.

For future builds, be sure to keep the cross-compilation build directory and your headers, and be sure to continue exporting the following variables (which you probably can throw into a script file):

```
set +h
umask 022
export LJOS=~/lj-os
export LC_ALL=POSIX
export PATH=${LJOS}/cross-tools/bin:/bin:/usr/bin
unset CFLAGS
unset CXXFLAGS
export LJOS_HOST=$(echo ${MACHTYPE} | sed "s/-[^-]*/-cross/")
export LJOS_TARGET=x86_64-unknown-linux-gnu
export LJOS_CPU=k8
export LJOS_ARCH=$(echo ${LJOS_TARGET} | sed -e 's/-.*//'
```

```
 ↪-e 's/i.86/i386/')
export LJOS_ENDIAN=little
export CC="${LJOS_TARGET}-gcc"
export CXX="${LJOS_TARGET}-g++"
export CPP="${LJOS_TARGET}-gcc -E"
export AR="${LJOS_TARGET}-ar"
export AS="${LJOS_TARGET}-as"
export LD="${LJOS_TARGET}-ld"
export RANLIB="${LJOS_TARGET}-ranlib"
export READELF="${LJOS_TARGET}-readelf"
export STRIP="${LJOS_TARGET}-strip"
```

Petros Koutoupis, *LJ* Contributing Editor, is currently a senior platform architect at IBM for its Cloud Object Storage division (formerly Cleversafe). He is also the creator and maintainer of the RapidDisk Project. Petros has worked in the data storage industry for well over a decade and has helped pioneer the many technologies unleashed in the wild today.

Building a Voice-Controlled Front End to IoT Devices

Apple, Google and Amazon are taking voice control to the next level. But can voice control be a DIY project? Turns out, it can. And, it isn't as hard as you might think.

By Michael J. Hammel

Siri, Alexa and Google Home can all translate voice commands into basic activities, especially if those activities involve nothing more than sharing digital files like music and movies. Integration with home automation is also possible, though perhaps not as simply as users might desire—at least, not yet.

Still, the idea of converting voice commands into actions is intriguing to the maker world. The offerings from the big three seem like magic in a box, but we all know it's just software and hardware. No magic here. If that's the case, one might ask how anyone could build magic boxes?

It turns out that, using only one online API and a number of freely available libraries, the process is not as complex as it might seem. This article covers the Jarvis project, a Java application for capturing audio, translating to text, extracting and executing commands and vocally responding to the user. It also explores the programming issues related to integrating these components for programmed results. That means there is no machine learning or neural networks involved. The end goal is to have a selection of key words cause a specific method to be called to perform an action.

APIs and Messaging

Jarvis started life several years ago as an experiment to see if voice control was possible in a DIY project. The first step was to determine what open-source support already existed. A couple weeks of research uncovered a number of possible projects in a variety of languages. This research is documented in a text document included in the docs/notes.txt file in the source repository. The final choice of a programming language was based on the selection of both a speech-to-text API and a natural language processor library.

Since Jarvis was experimental (it has since graduated to a tool in the IronMan project), it started with a requirement that it be as easy as possible to get working. Audio acquisition in Java is very straightforward and a bit simpler to use than in C or other languages. More important, once audio is collected, an API for converting it to text would be needed. The easiest API found for this was Google's Cloud Speech REST API. Since both audio collection and REST interfaces are fairly easy to handle in Java, it seemed that would be the likely choice of programming language for the project.

After audio was converted to text, the next step would be some kind of text analysis. That analysis is known as Natural Language Processing. The Apache OpenNLP library does just that, making it simple to break a text string into its component parts of speech. Since this was also a Java library, the selection of Java as the project language was complete.

Initially, the use of Google APIs included non-public interfaces—basically using interfaces hidden inside the Chrome browser. Those interfaces went away and were replaced by the current public Google Cloud Speech API. Additionally, Google's Translate text-to-speech feature was used, but that interface was removed after it was abused by the public. So an alternative solution was recently integrated: espeak-ng. Espeak is a command-line tool for speech synthesis. It can be integrated with voices from the mbrola project to help produce better (at least less computer-ish) voices. Think of it as an improved Stephen Hawking voice. Most important, espeak can be called directly from Java to generate audio files that Java then can play using the host's audio system.

With a set of APIs, tools and libraries in hand, it is now possible to design a program flow. Jarvis utilizes the following execution threads:

- Capture and record audio to a file.
- Convert the audio file to text.
- Process the text to parts of speech.
- Analyze the parts of speech for command processing.
- Provide a vocalized response.

Jarvis also includes a simple UI, mostly for debugging purposes, which can graph thread processing times and show voice patterns. Each thread operates with a short delay loop that checks for inbound messages from other threads. The recording

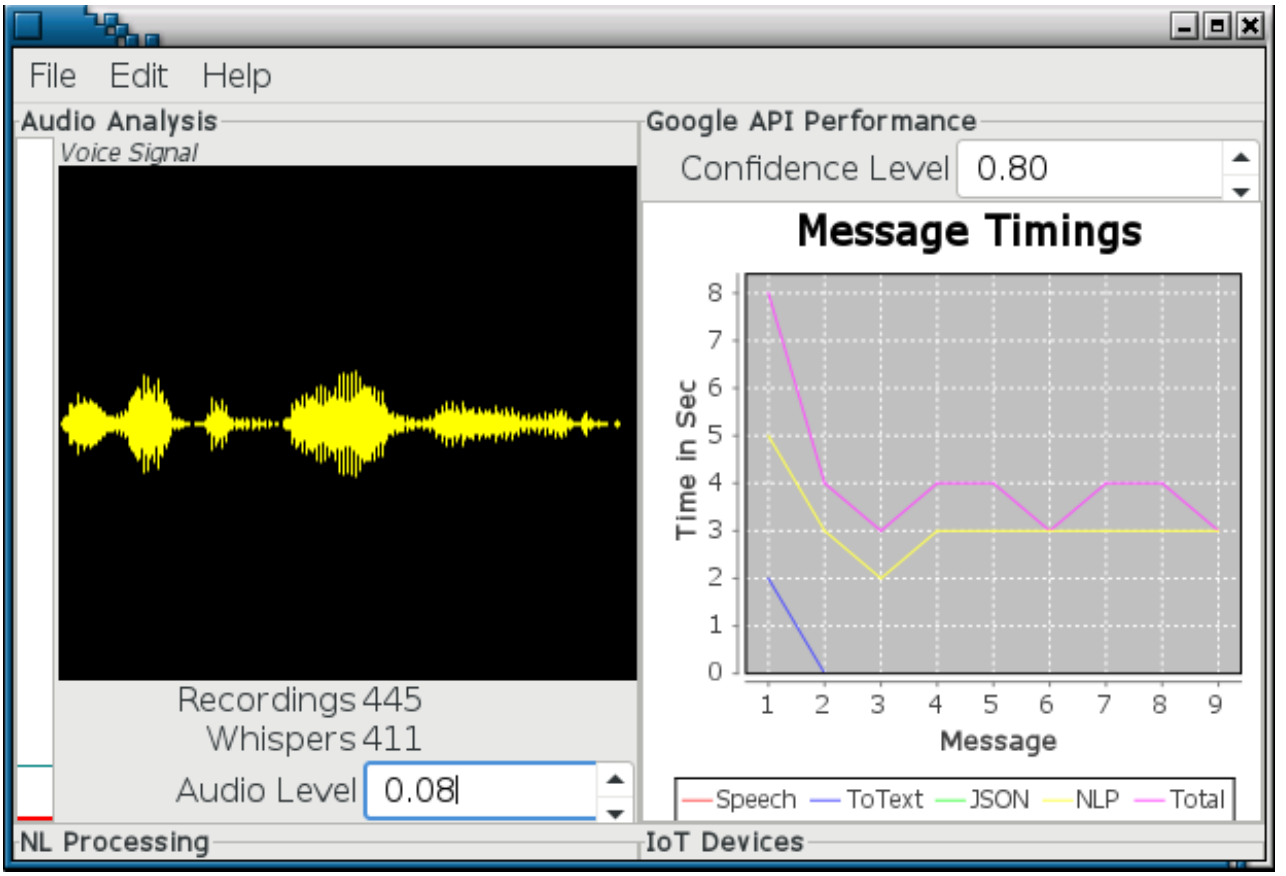


Figure 1. Jarvis' UI is primarily for monitoring processes and not directly for user interaction.

thread looks for audio above a configured level that is then saved to file when the level drops below a threshold value. This causes a message to be passed to the next thread for continued processing and so on. Even the UI thread is messaged with telemetry data to display.

Prerequisites

In order to use Jarvis, you will need a microphone and some speakers. The PulseAudio Plugin was used in testing because it allows enabling and disabling of the input source, which means I don't have the audio input device enabled all the time. No sense letting hackers hear my every move, should they get that far into my world.

Any microphone can be used, but for long-range development plans, where audio

should be picked up no matter where I am in a room, I picked up a Blue Snowball iCE Condenser Microphone. This is good quality for podcasts, and it does a fair job of picking up a clean recording of my voice anywhere in the room as long as I speak a little loud.

Speakers are necessary only if you want to hear Jarvis' responses. If you don't have any speakers for your computer, it won't prevent Jarvis from otherwise processing your voice commands.

From Speech to Text

Java provides support for reading from Linux audio subsystems through the `javax.sound.sampled.DataLine` and `javax.sound.sampled.AudioSystem` classes. The first step in using these is to set up an `AudioFormat` class with the required sample rate and associated configuration:

```
AudioFormat getAudioFormat() {
    float sampleRate = 16000.0F;
    int sampleSizeInBits = 16;
    int channels = 1;
    boolean signed = true;
    boolean bigEndian = false;
    return new AudioFormat(sampleRate, sampleSizeInBits,
     ↪channels, signed, bigEndian);
}
```

These settings are used later to determine the threshold level for sampling audio. `DataLine` is then passed this class to set up buffering audio data from the Linux audio subsystem. `AudioSystem` uses `DataLine` to get a Line object, which is the connection to the actual audio:

```
audioFormat = getAudioFormat();
DataLine.Info dataLineInfo = new DataLine.Info(
 ↪TargetDataLine.class, audioFormat);
```

```
line = (TargetDataLine) AudioSystem.getLine(dataLineInfo);
```

The line object is then opened, starting Java sampling of audio data. The line's buffer is tested for content, and if any is found, the audio level for that buffer is calculated. If the level is above a hard-coded threshold, audio recording is started. Recording writes the audio buffer to a stream buffer. When the level drops below the threshold, the recording is stopped, and the output stream is closed. This is called a snippet. If the snippet size is non-zero, you queue it for conversion to a WAV format using javaFLACEncoder:

```
int count = line.read(buffer, 0, buffer.length);
if ( count != 0 )
{
    float level = calculateLevel(buffer,0,0);
    if ( !recording )
    {
        if ( level >= threshold )
            recording++;
    }
    else
    {
        if ( level < threshold )
        {
            recording=0;
            Snippet snippet = new Snippet();
            snippet.setBytes( out.toByteArray() );
            if ( snippet.size() != 0 )
               snippets.add( snippet );
    }
}
```

Calculating the threshold requires running the length of the audio buffer to find a max integer value. Then the max value is normalized from 0.0 to 1.0, a percentage of the

maximum volume. The percentage is used to compare against the threshold level. The encoded WAV file is then queued for conversion to text.

Converting the audio file to text requires passing it through Google's Cloud Speech API, a REST API that requires an API key and has a financial cost, albeit a very low one (practically zero) for the average Jarvis user. Jarvis is designed to allow users to utilize their own key as one is not provided in the source code.

Google's API requires passing the audio file as Base64-encoded data in a JSON object in the body of an HTTP message, where the destination URL is the REST API. The return data is also a JSON object, filled with the converted text and additional data. Jarvis uses a custom class, GAPI, to hold the returned text data and handles JSON parsing to extract fields for use by other classes. The SimpleJSON library is used for all JSON manipulation:

```
Path path = Paths.get( audiofile.getPath() );
byte[] data = Base64.getEncoder().encode( Files.readAllBytes(path) );
audiofile.delete();
if ( (data != null) || (data.length.0) )
{
    String request = "https://speech.googleapis.com/v"
 ↪+ Cli.get(Cli.S_GAPIV) + "/speech:recognize?key=" + apikey;
    JSONObject config = new JSONObject();
    JSONObject audio  = new JSONObject();
    JSONObject parent = new JSONObject();
    config.put("encoding","FLAC");
    config.put("sampleRateHertz", new Integer(16000));
    config.put("languageCode", "en-US");
    audio.put("content", new String(data));
    parent.put("config", config);
    parent.put("audio", audio);

    ...use HttpURLConnection to POST the message...
```

```
    ...get response with a BufferedReader object...
    ...queue text from response for command processing...
}
```

From Text to Commands

Command processing forwards the text response to a GAPI object to determine what comes next. If Google found recognizable text, it is queued for conversion into parts of speech by the Apache OpenNLP library via Jarvis' NLP class. This tokenizes the text into collections of verbs, nouns, names and so forth. The NLP class is used to identify whether the command was intended for Jarvis (it must have contained that name):

```
private boolean forJarvis(NLP nlp)
{
    String[] words = nlp.getNames();
    if ( words == null )
        return false;
    for(int i=0; i<words.length; i++)
    {
        if ( words[i].equalsIgnoreCase("jarvis") )
            return true;
    }

    words = nlp.getNoun();
    if ( words == null )
        return false;
    for(int i=0; i<words.length; i++)
    {
        if ( words[i].equalsIgnoreCase("jarvis") )
            return true;
    }
    return false;
}
```

Requests meant for Jarvis are searched for key words in various formats and ordering to identify a command. The following code is used to respond to "Are you there", "Are you awake", "Do you hear me" or "Can you hear me" commands. The Java `String matches()` method is used to glob related keywords, making it easier to find variations on a command:

```
private boolean isAwake(NLP nlp)
{
    int i=0;
    String[] token = nlp.getTokens();
    boolean toJarvis = false;
    String[] tag = nlp.getTags();
    if ( tag != null )
    {
        String words = "are|do|can";
        for(i=1; i<tag.length; i++)
        {
            if ( tag[i].startsWith("PR") )
            {
                if ( token[i].equalsIgnoreCase("you") )
                {
                    if ( token[i-1].toLowerCase().
↪matches("("+words+")") )
                    {
                        toJarvis = true;
                        break;
                    }
                }
            }
        }
    }
    if ( !toJarvis )
        return false;
```

```
    token = nlp.getVerbs();
    if ( token != null )
    {
        String words = "listen|hear";
        for(i=0; i<token.length; i++)
        {
            if ( token[i].toLowerCase()
↪.matches("("+words+").*") )
                return true;
        }
    }

    token = nlp.getAdverbs();
    if ( token != null )
    {
        String words = "awake|there";
        for(i=0; i<token.length; i++)
        {
            if ( token[i].toLowerCase()
↪.matches("("+words+").*") )
                return true;
        }
    }
    return false;
}
```

Once a command is found, it can be turned into an action. This part of Jarvis is still evolving, but the intent will be to use a REST API to contact a PiBox server with the command. The PiBox server will be responsible for contacting IoT devices like light switches or window shades to perform the appropriate action.

This type of programmed response is highly inefficient. Sequential processing of

commands is slow and will only get worse with more commands. However, it serves as a reasonable implementation for simple support of home-automation commands.

Giving Voice to Your Creation

After the action is processed (or queued on the remote PiBox server for handling), a vocalized response can be queued. In the above example, the response is "Yes, I'm here. Can I help you?" This text is placed in one of Jarvis' internal Message objects to be queued on the `Speaker` thread.

Text is extracted from the `Message` object, and a command line is built using the `espeak-ng` utility. Espeak is a package you can install from your Linux distribution package management system. On Fedora, the command is:

```
sudo dnf install espeak-ng
```

Note that there is the original `espeak` and the rebooted `espeak-ng`. They seem to produce the same results for Jarvis and have the same command name (`espeak`), so either should work.

The `espeak` program takes text as input and outputs an audio file. That file is then read and played using Java's sound support:

```
String cmd = "espeak-ng -z -k40 -l1 -g0.8 -p 78 -s 215 -v mb-us1
 ↪-f " + messageFilename + " -w " + audioFilename;
Utils.runCmd(cmd);

file = new File(audioFilename);
InputStream in = new FileInputStream(audioFilename);
AudioStream audioStream = new AudioStream(in);
AudioPlayer.player.start(audioStream);
file.delete();
```

In this example, the `runCmd()` method is a Jarvis wrapper around Java process

management to simplify running an external command.

Caveats

Jarvis relies on Google's Cloud Speech API to convert audio files to text. This API also requires an API key provided by Google. Since this service is not free, Jarvis allows developers to place their own API key in the docs directory and the build system will find it. Alternatively, if you run Jarvis without rebuilding, just place the `apikey` in the ~/.jarvis directory.

This DIY system is not completely secure. The use of the Google Cloud Speech API implies the transfer of audio files across the internet to Google, which translates it to text. This means that Google has access to the audio, the converted text and the source of both. If you are concerned with privacy, be sure to consider this issue before using Jarvis.

The source code in this article is simplified from actual Jarvis code for the sake of explanation only. Note that the Jarvis source code, although written in Java, is not designed for building with Maven or within Eclipse. The build system has been crafted manually and is intended to be run from the command line with Ant. Use `ant jarvis` to build, `ant jarvis.run` to run from the build artifacts and `ant rpm` to generate an RPM file if used on Fedora, Red Hat or CentOS systems. See `ant -p` for a list of supported targets. Also note that Jarvis is a Java program written by a C developer. Caveat programmer.

Going Forward: AI

Command processing in Jarvis is programmed, which means there is no AI here. Commands are parsed from text and grouped loosely to allow for variations in commands, such as "are you", "can you" and "do you" all referring to the same command processing track.

This mechanism should be expanded to support a configuration language so that commands can be extended without having to rewrite and extend the code or recompile. Ideas related to this are being considered currently but no design or

implementation has started.

Taking Jarvis beyond mere programmed responses requires integration with an AI back end. The TensorFlow project from Google offers a Java API and is a logical next step. However, for basic home automation, the integration of AI is a bit of over-engineering. There currently are no plans for TensorFlow or other AI integration in Jarvis.

Jarvis is still just an experiment and has plenty of limitations. Message flow through a series of threads slows processing significantly and shows up as delays in responses and actions. Still, Jarvis' future is bright in my home. The IronMan project will integrate Jarvis with a PiBox server for distributing commands to IoT devices. Soon, I'll be able to walk into my office and say "Lights on, Jarvis". And that's a very bright idea, indeed. ■

Michael J. Hammel is a Software Engineer for NetApp living with his wife Brinda and two Golden Retrievers in Broomfield, Colorado. When he isn't working on embedded systems or other geekery, he likes to camp, walk his dogs around the park, and drink tea with his wife and revel in the joy of his daughter's success. He has written more than 100 articles for numerous online and print magazines, and he's the author of four books on GIMP, the GNU Image Manipulation Program.

Open Hardware: Good for Your Brand, Good for Your Bottom Line

With the rise of IoT, we're inside a short window where "open" is a strong differentiator for hardware products. Is your company ready to take advantage of it?

By VM (Vicky) Brasseur

I don't know how to put this, but Hardware is kind of a Big Deal, and thanks to the Internet of Things (aka *IoT*), it's getting bigger every year. The analyst firm IDC expects spending on IoT to reach nearly $800 billion USD by the end of 2018. A study by Intel shows that by 2025, the global worth of IoT technology might be as high as more than $6 trillion USD; whereas Forbes reports that the global market could be nearly $9 trillion USD in 2020.

These statistics are based on the traditional model of closed design and development of the chips, boards and objects that will make these devices a reality. However, what if hardware developers were to learn from and leverage the popularity of free and open-source software (aka *FOSS*)? What if the future of IoT were Open? It's my belief that the device developers who apply the lessons of FOSS to hardware development will be those best positioned to become the powerhouses of that $9 trillion market. Similarly to software, open hardware will be seen first as a differentiator (rather than an eccentricity) and later, as the industry matures, as the default operating mode for hardware development.

Open hardware will require a lot of evolution before it becomes the default

development paradigm for IoT. The current state of open hardware closely resembles that of free software 30 years ago: it exists, but in a nebulous form. While organisations like the Open Source Hardware Association and the Open Hardware Repository are working to change this, the fact remains that there currently are few clear legal, social or procedural norms around open hardware development. There is no hardware equivalent of the Open Source Initiative's list of approved licenses or agreement on whether a license is even needed at all. The diversity of tools required for hardware design makes collaboration and interoperability more difficult. These legal and procedural questions contribute to a very high barrier to entry for releasing designs as open hardware. That barrier isn't insurmountable, however. Already advancements in 3D printing as well as emulation and virtualization through digital twins are enabling types of collaboration that were impossible just a few short years ago. As these methods continue to adapt, the resulting emergent and open processes and norms will drive further and faster innovations.

The openness of hardware also will be hampered for a while by the limitations imposed by chip makers. For instance, the Mark I voice assistant speaker is certified as open hardware, but it includes components built upon proprietary chipsets and boards. Joshua Montgomery, CEO of Mycroft, sees this as a valuable stepping stone toward an entirely open hardware platform. "For now, a lot of open hardware is like LEGO: while the bricks themselves aren't open, the designs built with them can be." This stepping stone has enabled Mycroft to iterate on the Mark I, which incorporates off-the-shelf hardware like Raspberry Pi and Arduino, to create the Mark II, which features a board custom- and openly-designed for security, privacy and flexibility.

Chip makers are starting to catch on to the advantages of open, however. SiFive has released an entirely open RISC-V development board. Its campaign on the Crowd Supply crowd-funding website very quickly raised more than $140,000 USD. The board itself is hailed as a game-changer in the world of hardware. Developments like these will ensure that it won't be long before the hardware equivalent of LEGO's bricks will soon be as open as the designs built using them.

In the March 2018 issue of *Linux Journal*, I encouraged companies to wait until

they're secure and established in their market before releasing mission-critical software as open source. The reason here is, of course, that it's quite simple for someone else to take that software, spin up a new cloud instance and create a company that competes in your market. There are very good business reasons for a company retaining its software intellectual property until the company has locked in its market.

However, that's not currently the case for hardware. The complexities involved with manufacturing are far more daunting than those around launching a new software firm. The knowledge, regulations, testing and tooling involved with bringing new hardware to market create a massive—and massively protective—barrier that's compounded by those undefined legal, social and procedural norms I mentioned earlier. This provides open hardware startups the freedom to share their designs after (or even before) product launch while still retaining prime mover advantage, simply because no other company would be able to spin up the tooling required quickly enough to impact the sales and market of that prime mover. Once a piece of open hardware is released to customers, the company creating it is already so far ahead of any others that would use its designs that there is little danger of market loss or competition. According to Josh Lifton, CEO of Crowd Supply, even after helping hundreds of creators launch their open hardware products on the platform, "...we have never gotten any report of an idea from a Crowd Supply project getting stolen."

While open hardware does not share free and open-source software's business risks for releasing of intellectual property, it does share its difficulties in attracting contributors and building a community. While both open endeavors represent the field of dreams for creators, it remains true that if you build it, they will *not necessarily* come. This is especially true for open hardware where people are not only less used to the idea of contributing, they're also often less prepared to do so. Aside from the specialized tooling required to read, manipulate and understand hardware design files, there's the additional problem of a knowledge gap. Relatively speaking, far more people are equipped with the knowledge and resources necessary to contribute to FOSS projects. When source files are distributed for a free and open-source

software project, many people already know what to do with them. When source files are distributed for an open hardware project, comparatively few people will have the electrical engineering, industrial design, stress analysis or other related experience (let alone the very expensive tools required) to contribute effectively. While this naturally leads to considerably fewer contributions to and collaborators on an open hardware project, it also means that those collaborators are more likely to be qualified and their contributions of a higher quality.

Hope of contributions should not be the primary reason for releasing hardware designs under an open license. As Kaia Dekker, CEO of Keyboardio points out, building a business while relying on volunteers for contributions "really isn't something that necessarily works out". This is a consideration that's overlooked by a lot of the new breed of "open first" software businesses springing up these days, but that the sparse contributions make impossible to ignore for open hardware.

Despite this, open hardware still allows companies to reduce the time and expense required for research and development. For instance, Dekker says, developing the Keyboardio Model 01 would not have been practical without the open platform of Arduino or the vibrant community around it. Open hardware components are similar to libraries or modules for software: they allow an organisation to add functionality without having to develop it themselves. Entirely new products, such as the Keyboardio Model 01 or the Mycroft Mark I, can come to market thanks to what are essentially off-the-shelf open hardware components.

Joshua Montgomery of Mycroft AI points out that as important as the R&D benefits were to the company when they were getting started, they're finding that opening the new design of the Mark II is also opening possibilities that they'd not have seen had they kept their hardware proprietary. Through careful design of the Mark II board, Mycroft has created a new off-the-shelf component that partners and other businesses can use, in combination with the open-source Mycroft Artificial Intelligence platform, to add a voice assistant feature to their own devices. This increases adoption of the AI platform software along with the contributions and collaborators on it.

By designing their hardware to work in many different contexts, then releasing it as an open solution, Mycroft also has gained the benefit of economy of scale for the company as well as for all companies using the open component. When all companies order the component from the same manufacturer, the subsequent "run" of the part becomes cheaper for every company that uses it. This open supply chain also reduces risk for all participants. Should a supplier fail or withdraw from manufacturing a component, with open hardware, it's possible to spin up tooling in another location. With proprietary hardware, it's possible, if not likely, that companies reliant upon that component will need to redesign their products when a component is no longer available. Naturally this type of economy of scale will depend a lot upon the market and company strategy, but the Mycroft example, even in these early days of open hardware, proves that the open approach can provide innovation opportunities and benefits to innumerable companies and entrepreneurs.

According to Kaia Dekker, none of these benefits would be possible without the communities that form around open hardware projects. "It opens doors to us that would be closed otherwise." Keyboardio has found unexpected success in recruiting firmware development and other talent from its community. That same community can develop after-market accessories for the Model 01, adding, if they wish, features such as wireless capability, thanks to the open hardware at the core of the device. This ecosystem of accessories as well as the supportive community has strengthened the Keyboardio brand as its community members have become their most passionate and outspoken advocates. Keyboardio has found that the transparency has increased its perceived reliability in the eyes of its target market, which itself increases the market's trust in the company. Josh Lifton agrees and says that many Crowd Supply creators have seen similar results. "A lot of these people are early adopters and innovators. This type of person usually has a strong community-minded ideology."

The transparency of open hardware does more than build trust in its communities. It also builds trust in the entire customer base. According to Joshua Montgomery of Mycroft, the "radical transparency" of the Mark I and Mark II devices help to support their reputation for privacy and security. Increasingly, this sort of reputation can have a huge impact on companies looking to enter certain markets. Open

hardware ensures that there is never a black box. Consumers, regulators and security auditors all have equal access to the designs of the devices. In an environment where customers are more privacy-aware than ever, this type of transparency can be a powerful differentiator for products.

In 1997, Clayton Christensen detailed the dangers of ignoring the new paradigms in business in *The Innovator's Dilemma*. Hardware and IoT creators today are at risk of stumbling straight into that dilemma if they don't embrace the type of innovation and collaboration that free and open-source software has proved is possible. While the ecosystem and contribution norms are still taking shape around open hardware, it's clear from the benefits listed here that the current environment provides enormous opportunities for companies willing and able to work with and mold that ecosystem. The companies that recognise this now are those that will lead the way for several decades to come. ■

VM (aka Vicky) Brasseur spent most of her 20 years in the tech industry leading software development departments and teams, and providing technical management and leadership consulting for small and medium businesses. Now she leverages nearly 30 years of free and open-source software experience and a strong business background to advise companies about free/open source, technology, community, business and the intersections between them. She is the author of *Forge Your Future with Open Source*, the first book to detail how to contribute to free and open-source software projects. Think of it as the missing manual of open-source contributions and community participation. The book is published by The Pragmatic Programmers and is now available in an early release beta version at https://fossforge.com. Vicky is the proud winner of the Perl White Camel Award (2014) and the O'Reilly Open Source Award (2016). She's a moderator and author for opensource.com, a Director for the Open Source Initiative, and a frequent and popular speaker at free/open-source conferences and events. She blogs about free/open source, business and technical management at http://anonymoushash.vmbrasseur.com.

Linux Gets Loud

Exploring the current state of musical Linux with interviews of developers of popular packages.

By Joshua Curry

Linux is ready for prime time when it comes to music production. New offerings from Linux audio developers are pushing creative and technical boundaries. And, with the maturity of the Linux desktop and growth of standards-based hardware setups, making music with Linux has never been easier.

Linux always has had a place for musicians looking for inexpensive rigs to record and create music, but historically, it's been a pain to maintain. Digging through arcane documentation and deciphering man pages is not something that interests many musicians.

Loading up Linux is not as intimidating as it once was, and a helpful community is going strong. Beyond tinkering types looking for cheap beats, users range in experience and skill. Linux is still the underdog when it comes to its reputation for thin creative applications though.

Recently, musically inclined Linux developers have turned out a variety of new and updated software packages for both production and creative uses. From full-fledged DAWs (Digital Audio Workstations), to robust soft-synths and versatile effects platforms, the OSS audio ecosystem is healthy.

A surge in technology-focused academic music programs has brought a fresh crop of software-savvy musicians into the fold. The modular synth movement also has nurtured an interest in how sound is made and encouraged curiosity about the technology behind it.

One of the biggest hurdles in the past was the lack of core drivers for the wide variety of outboard gear used by music producers. With USB 2.0 and improvements in ALSA and JACK, more hardware became available for use. Companies slowly have opened their systems to third-party developers, allowing more low-level drivers to be built.

Hardware

In terms of raw horsepower, the ubiquity of multicore processors and cheap RAM has enabled Linux to take advantage of powerful machines. Specifically, multithreaded software design available to developers in the Linux kernel offer audio packages that offload DSP and UI to various cores. Beyond OS multithreading, music software devs have taken advantage of this in a variety of ways.

A well known API called Jack Audio Connection Kit (JACK) handles multiple inter-application connections as well as audio hardware communication with a multithreaded approach, enabling low latency with both audio DSP and MIDI connections.

Ardour has leveraged multithreaded processing for some time. In early versions, it was used to distribute audio processing and the main interface and OS interaction to separate cores. Now it offers powerful parallel rendering on a multitude of tracks with complex effects.

In addition to core advantages, peripheral compatibility has greatly improved. Linux audio had a reputation for a lack of drivers for legacy outboard hardware. Much of that was built around PCI and Firewire hardware that often was released with closed APIs, and Linux compatibility was an afterthought, if considered at all.

With the advent of USB 2.0, a whole ecosystem of external interfaces became available. Drivers were still needed for specific chipsets, but the actual connection was now negotiated by a widely adopted standard.

In addition, Advanced Linux Sound Architecture (ALSA) has continued to mature. Its integration into Linux kernel 2.6 back in 2003 established a commitment to useful and

easy-to-use abstraction of a large collection of audio drivers. ALSA 1.1.5 now boasts complete or partial compatibility with audio hardware from more than 120 vendors, encompassing hundreds of implementations.

A relatively new means of music machine communication, Open Sound Control (OSC), has emerged as a significant alternative to MIDI. Commonly sent over UDP/IP, it offers more flexibility than the hardware-bound MIDI. As an open content format, it offers diverse data types and the ability to integrate messaging beyond musical signifiers.

Although not specific to Linux, it has been embraced by Linux audio developers and many popular packages offer some kind of OSC integration. Notably, software synthesizer ZynAddSubFx has gone completely to OSC internally while still offering MIDI compatibility.

The attention to performance and efficiency that Linux has enjoyed now finds a new application. It's well known that Linux performs well even on older machines. The surge in popularity of miniaturized Single Board Computers (SBCs) like the Raspberry Pi has brought fresh attention to the capabilities of Linux.

Interest in headless environments for audio has taken advantage of lean technologies built in to the kernel. Many new DIY synthesizers, audio routers, sensor-based audio generators, localized performance networks and MIDI controllers are all based on SBCs running Linux.

Ready for Real Time

Crucial to musical performance is the need for immediate feedback from instruments. The latency between a keyboard trigger and its sound needs to be as low as possible to retain feeling in a performance.

With slower PC hardware, efforts to speed up the DSP layers and MIDI message paths led to a community effort to find the leanest environments possible and develop real-time kernel builds.

Real-time priority for processes leads to trade-offs in OS performance and stability, so some approaches come at a usability cost. But, with modern multicore processors running at high speeds, the need for real-time kernels is mitigated.

An informal survey from linuxmusicians.com reports that most users opt for a lean distro, such as Arch Linux. A popular preconfigured meta-package is KXStudio, started by Philipe Coelho, who describes its creation:

> The KXStudio project began as I started to settle down on a single distribution and wanted the latest updates for a few packages, but without having to run unstable or rolling-release distributions. I was just quickly trying to package things, then putting everything together into an ISO file.

He recently has pivoted to focus more on the repositories and plugins though. Cadence, his collection of Linux audio utilities for JACK and much more, is regularly cited as a must-have for Linux audio. There is still a live DVD image of the original KXStudio available for users wanting to get a sense of a mainstream audio setup on their own systems.

Regarding real-time kernels Coelho offered:

> It's not as important as it used to be. But it's something worse pursuing and trying if one is going to record multiple streams of audio concurrently. Sequencing a song where the result is rendered "offline" does not require a real-time kernel at all. The way to go (and the easiest) is to try to do your music without a real-time kernel until you run into limitations. If it gets to a point where you need one, go for it.

DAWesome

For studio music production, a Digital Audio Workstation (DAW) is an essential tool. It's the captain's bridge from which a complete workflow of music recording, editing, creation and rendering is commanded. Linux has excellent options for professional audio work.

Ardour has been around since 2005 and has a solid reputation. Now at version 5.1, it offers a full suite of multitrack and multiplatform audio production capabilities. It has no limits on the number of tracks or plugins. If your machine can handle it, Ardour can wrangle it.

It offers an audio interface decoupled from JACK, but retains low latency. Multicore-friendly since its inception, it offers the ability to dedicate specific numbers of cores to DSP and GUI operations. You can record in layered, non-layered or destructive modes on a per-track basis. A pioneering feature is unlimited undo/redo, even across saved sessions.

Linux Multimedia Studio (LMMS) is comparable to the venerable Fruity Loops Studio or even Apple's Garageband. It also enjoys the support of many active Linux audio developers, with more than 130 contributors to its GitHub repo.

Geared toward song crafting in a unified interface, it offers "phrase" arrangement for verse/chorus/verse and pattern blocks. In addition to outputting to MIDI instruments, it comes with a quiver of soft synths and plugins for diverse tastes. Choose from 16 built-in synthesizers, including emulations of Roland TB-303, Commodore 64 SID microchip and the Yamaha OPL2 chip. The powerful ZynAddSubFx also is included with deep integration.

Using a different approach to sequencing, Renoise is a commercial ($75) offering with a deep feature set. Interestingly, Renoise is a "tracker" under the hood. People might remember the MOD trackers from the 1990s, descendants of the 1987 Amiga release Ultimate Soundtracker. Trackers have continued to evolve since then, and they offer a distinct approach to step-sequencing and grid pattern arrangement.

Renoise 3.1 is a significantly advanced implementation of the tracker concept, with a newly revamped sampler and slice arranger and compatibility with a huge array of virtual instruments and plugins. EDM producers and beat crafters love Renoise, and it has a thriving user community.

Climbing up on the cost scale is Bitwig Studio at $399. With a focus on arrangements and live performance, Bitwig is a fast and versatile production tool. It looks amazing, and the UI polish allows for intuitive engagement with the platform.

Effect and modulator chaining is arranged in a novel container format called "Device Nesting". For sound designers seeking complex chains with full automation, this is a standout feature.

Bitwig is also one of the few DAWs that supports Multidimensional Polyphonic Expression (MPE). MPE is used in a new generation of MIDI controllers like the Roli Seaboard. In addition to standard MIDI expressions like velocity, MPE offers multichannel data streams from vibrato, timbre, layer triggering and much more. Think violin over piano.

Many more Linux DAW packages are available, such as MusE, Radium, Qtractor and Rosegarden. A useful comparison matrix of all of these (created by user "milk") is available at linuxmusicians.com.

Helm's Deep

Standalone FLOSS synths are getting more uncommon these days. A new entry into the club is Helm, an interesting and stable synthesizer with an intuitive layout and clean UI. It also comes as a cross-platform LV2, VST, VST3 or AU plugin in both 32-bit and 64-bit versions, all under a GPL license. Released in 2015, it has rapidly improved and is now at v0.9.0.

Besides dual oscillators, sub octave and noise generators, it offers distinctive stutter and formant controls. LFOs can be routed to parameter changes across modules as well.

A fan of Helm is Mark McCurry, maintainer of the ZynAddSubFx project. "It has great routing and great design", he says.

McCurry has been busy himself these days with the release of Zyn-Fusion. It is a

Figure 1. Helm

complete overhaul of the ZynAddSubFx UI, reducing 28 separate control windows to a single master window pane. Along with improvements to ZynAddSubFx itself, now at 3.0.3, Zyn-Fusion brings the venerable synth package into the modern age. According to McCurry:

> In mid-2014 a series of mockups for a new UI for Zyn were proposed. At that point this was a yearly event. Someone proposed a GUI, using it would break a majority of the workflow of the app or it was just a glorified beginner mode. Then someone would insist on trying to implement 1/10th of the mockup after which point it was abandoned.

Figure 2. Zyn-Fusion

> The mockup by Budislav Stepanov, however, was different. It had all of the controls, it provided a much better organization, and it looked good.
>
> I figured the only way to ever see that GUI materialized was to just do it and funding would make it realistic.
>
> I took a look at snowdrift's research on funding models and took a look at the struggles that openAV had with their funding. This information was used to decide upon the funding model.

The initial release of Zyn-Fusion was priced at a reasonable $45, and McCurry states that the response was positive. As planned, the new UI has now been open-sourced and no longer requires purchase.

Another project gaining steam is the modular synth platform VCV Rack. Taking a cue from the surging popularity of the hardware-based Eurorack modular

synthesizer format, VCV Rack is robust software implementation of a patch cord-based modular synthesizer.

Boasting a rich array of modulators, filters, oscillators and audio generators, VCV Rack is capable of creating very complex instruments. The UI is very clean, and attention to detail is impressive.

Currently free under the BSD-3-Clause license, core developer Andrew Belt explains:

> I treat VCV Rack as a commercial project in all respects except for the price and license. Instead of selling Rack, I give it away for free to increase its popularity a hundred-fold while selling VCV-branded plugins which support my time spent on the Rack platform. Instead of restricting Rack with a freeware proprietary license, it is licensed under BSD for two reasons: I want to allow commercial and non-commercial plugins to include significant parts of Rack's DSP, graphics, and plugin framework into their software with no worries of licensing. I also want to allow users to have control over the software they use, by having the ability to review, study, and change the software if needed.

Belt shared his inspiration for the project:

> There is a similarity between modular synthesis and programming that I'm sure many programmers have noticed. Creating a mainstream polysynth VST plugin requires an experienced plugin developer with a well-designed plan, but modules for a virtual modular synthesizer nearly write themselves and automatically have great usability due to the same principle as the UNIX philosophy—many minimal but polished utilities combined into a large system yield better results than a monolithic system.

The Future of Linux Audio Development

With the rise of music-sharing sites like SoundCloud and streaming services like Spotify, the ability for musicians to get their music to the world has never been easier. The number of DIY music producers has grown exponentially. Interest in low-cost and DRM-free tools is at an all-time high.

Belt sees a movement toward more open software:

> If you're a programmer and not scared of reading manuals, Linux is a fantastic platform for experimentation in the intersection of music and creative coding. Linux works great for standalone software like VCV Rack, mod trackers and audio programming languages, but the same is true for Windows and Mac. From a "mainstream user" perspective, however, Linux is seen as hard to use with sparse support for hardware and lacking in software choices if you just want to "get things done", and I can't blame them. On the other hand, a possible rectifying factor for Linux is the increased privacy concern of Windows and Mac, loss of control for things like automatic updates and the unfortunate blurring of the line between the internet and your desktop. People already have started seeking refuge in a Linux distro of their choice to avoid these problems.

McCurry encourages people to contribute back to the community, "for people who want to get involved, my response is 'Just get started.'" Help is needed from all sides. Developers are sorely needed; documenters are needed; designers are needed; marketers are needed; users with well thought out critical feedback are needed. McCurry adds:

> For people just getting started, shop around. By looking at the mailing lists, forum posts and issue trackers for a project, you can get a feel for what's missing, what's being worked on and where things are going. Once you have a feeling for what you want to learn or contribute, start small. It's easy to get overwhelmed, and by taking small steps it's possible to go from knowing virtually nothing about a subject to being proficient.

Coelho echoes a similar sentiment:

> Best thing you can do first is to get yourself familiarized with Linux itself. Trying to "fix" things or develop something new without knowing what already exists and how it works can end up badly. We already have enough software that tries to do the same thing over and over again, but not reaching a really good usable state.

I think the biggest thing we're missing in Linux audio is documentation, and people willing to write more of it. I can speak for myself that I hate writing documentation—even for my own applications.

Good designers are missing too, but it's hard to find people with the right mentality. Good designers willing to put their work on free and open software are very rare.

Coda

With modern hardware and robust offerings from the Linux software community, a full music production studio can be built around the Linux platform. From DIY bedroom producers to mainstream artists, such as Deadmau5, music makers of all stripes are exploring the versatile and powerful audio capabilities that Linux has to offer. ■

Joshua Curry is a multimedia artist and musician living in San Jose, CA. He produces under the name Lucidbeaming. You can find him via his website, https://lucidbeaming.net or follow him on Twitter @lucidbeaming.

Enter Jakarta EE: an Inoculation Against Fear, Uncertainty and Doubt in the Java Community

Why I stopped worrying and learned to love changes in governance and branding.

By Dennis Gesker

Developers can be passionate about the tools and languages they use for development. This passion is a double-edged knife. It can foster growth of the technology's adoption and inspire the direction of energy into the language that one has chosen to advocate. The passion might also scare off those who wish to use the language or are just entering the field, particularly when the opposing view is exaggerated, incorrect or out of date with the current state of the technology. This latter scenario injects (often unintentionally) into the dialogue regarding the technology in question Fear, Uncertainty and Doubt (FUD).

Java Recap

Java was introduced in the mid-1990s. Issues inherent to the technology and the governance of the technology have been numerous through the years. Many of the issues raised have been valid and significant. Others, not so much. There have been issues with speed, floating-point arithmetic, handling of unsigned numbers and so

on. Most of those technical issues have been addressed as the language matured. Java recall is a catch-all for the base language, facilities that support the language (the JVM), its licensing and brand management in general. Also, the legal kerfuffle between Oracle and Google being notable and having spanned a number of years certainly opened the door for both legitimate and exaggerated FUD.

Many of the above performance and security issues have been addressed by software engineers in maintenance and full revision development releases. Indeed, the language continues to evolve, and even as version 8 of the platform seems to have hit its stride, version 9 reached general availability this past September, and the community of developers that rely on this language and facilities seem anecdotally well into experimentation and adoption. With version 9 still a pretty fresh and current release, version 10 already has reached its release candidate stage. Some Java issues have been addressed as the governance model evolved, legal issues resolved and license issues clarified. It did seem for a while various projects could have led to a fracture: IcedTea, Harmony and so on. However, these big players actively supporting and backing the OpenJDK project, bringing their side efforts, engineers and brand prestige with them, deserve a lot of credit for the acceleration and advancement of Java SE in recent years.

Recent Movement

One area that has been a recent hotbed of movement, discussion and, yes, a source for the generation of FUD, is the Enterprise Edition of Java. So, just what is all the hubbub?

In a nutshell, Oracle pushed out control of the Enterprise Edition—which for non-Java developers and discussion purposes here—can be thought of as all the "heavy duty"-level facilities in the Java toolbox not provided by the base language, JVM and JDK. Or, put another way, EE provides to us mere mortal and all-too-busy developers all manner of facilities, frameworks and other useful tools to abstract away quite a bit of the drudgery that goes with building enterprise-scale software routines.

Oracle's move on this standard and code base governance was a huge shift.

The "Java Way", Both Standard and Enterprise

For most of us Java folks, not a lot of time is spent thinking about memory management. Is this critical aspect of programming gone? No. But, an awful lot of boilerplate and routine tasks are simply taken care of by the facilities provided by the JVM. This catch-all "Java Toolbox" is taking care of most and often all memory management business for us.

I recall feeling a bit unsettled when I wrote my first Java program that I had declared and initialized variables and didn't circle back to clean up after myself and recover that memory. But, hey, the JVM was on it, and through the years, it grew more efficient at recovering resources no longer needed by the program. Dealing with memory was pretty much abstracted away. What was a strange sense of loss way back then seems almost intrusive when it comes up in a project now: "Hey, why am I even thinking about that? Okay, I'll do some JVM-tuning or optimize some code a bit."

In many ways, this abstracting just rolled on—sometimes faster than I realized. Some approaches stuck and made the cut into some layer of the Java SE/EE standards. Others fell from favor. Still, other fine libraries that were very well done and a delight to use (looking at you JODA time—JSR-310) seemed to be overlooked for a long time. Java SE, and to a greater extent Java EE, reflects this constant evolution a great deal. The evolutionary tree has lots of branches.

Ultimately, most included layers were beneficial. Developing routines to stitch together data from a variety of databases from a number of vendors is a typical programming task—just ordinary integration and reporting. Early on, it was great to be able to ride existing ODBC with JDBC connections! Soon after, JDBC just evolved, and the once-needed ODBC layer was discarded and unnecessary for the standard. It didn't happen overnight, but it happened. Eventually, it was apparent that Java developers still were writing quite a bit of custom or native SQL to get the data we needed and still dealing with differences across database products.

Behold, JPA—yet another layer of abstraction above JDBC—appeared. Now, in many ways, most database interactions are via the perspective of Java Objects and an entity

manager to deal with flipping data in and out of Java and various data stores like PostgreSQL or MS-SQL and even NoSQL databases like Mongo. The EE JPA facilities deal with many of the idiosyncrasies between vendor implementations. Often one needs only to let the EE facilities (embodied in a container) know the "flavor" or even the version of the RDMS(s) with which I need to communicate. Just as in the earlier example of Java and memory management above, it's not that I don't think about optimizations or picking up performance when working with databases—there are tools for when I need to, or I can bypass a layer for a particular use case—but a lot of the drudge work is just handed off to the provided facilities. I step in where needed. And, in situations where I engineer on the fly (too often I admit), I just add an optimization loop (usually in code review) to pick up the speed I'd like to have or back-log the optimization for a future dot release.

Java EE Drawbacks

Well, you have many layers and frameworks from which to choose, so you have to make decisions. There were times in the past when I would have addressed a database issue with a trigger or stored procedure. I still might in some cases today. But, before I do make a choice, I ask myself questions similar to "Is it worth it?" "Will the facility or abstraction provided get the job done?" And, of course, the very important, "Will staying in the container make it easier for the programmer that follows me or even a current teammate versus a performance pickup that might not be noticed if I optimize outside the Java provided (Standard or Enterprise) facilities?"

Of course, as with any mature language, there is a full ecosystem of projects that can fill in gaps not yet absorbed into the standard.

Java EE Upside

If a particular implementation of these extensions gets shaky—which is to say FUD enters the picture—take your code and jump to a different container. Why, because it became apparent to Sun Microsystems (acquired by Oracle) that it had a tiger by the tail and some kind of governance model would be needed to keep this Java creation on track. Was/is this model always good or smooth? No. But, did it provide a way to get folks involved? Yes.

From a big-picture point of view, it was a good call. Could it have made a bigger call? Say, push the Java out to the ISO folks or maybe the IEEE folks? Maybe. But, why debate hypotheticals, right? The point is there was a forum. The big picture is that warts, bumps and bruises aside of the governance approach the "facilities" that a Java developer can depend upon both at the JVM Standard/Base level and at the Enterprise/Extended level now had a pretty good level of predictability. If one EE product—or even a specific layer within a given EE product or container—gets shaky, just jump to one of the other 20 products that complies with this very capable lowest-common-denominator standard.

As Oracle began to signal that it was getting ready to find a new home for EE, it appeared that the reference implementation, Glassfish, was in jeopardy. What kind of jeopardy? Those of us that code against the reference implementation might not be able to get commercial support should we need it! Many developers didn't know which way the wind was going to blow, and uncertainty set in, which engendered in me a sense of fear in advance of going into production as the vendor we'd expect to turn to if trouble of some kind was encountered was seemingly but not definitively bailing on the standard code base it oversaw introducing doubt. It was the FUD trifecta with no help needed from the opinion leaders.

I depended on Glassfish. I didn't know what to do next. It was a truly terrifying six minutes! (The six minutes included refilling my coffee mug.)

I Googled Glassfish alternatives. JBoss was the first commercial link I hit. Wildfly was the upstream open-source and compliant base I hadn't re-visited in years. Hazaah! I had a path and wasn't painted into a corner. If I had scrolled down the page half an inch, I probably would have ended up taking Paraya for a spin instead.

The Take Away

In a decoupled polyglot programming world with EE at the core, a developer is not locked in with the logic routines coded. Precious coding hours are protected. There are plenty of open-source options, so development shops both large and small can tool up and develop on the cheap. There are lots of commercial and cloud

alternatives, so production launch stress is reduced. In addition, the EE container level frees one to try different outside products (storage, database, SAAS of the Month, web) at different price/performance points and quickly, because mostly it is a matter of configuring the container to point at these services and get back to the business of coding—all benefits of open source in general but some extra sense of safety from a standardized governed stack.

A governed standardized core stack like what is provided by Java SE/EE does not prevent one from stitching together some kind of polyglot solution. So, yes, one can play with the most current programming fad or marketing term of the week and if need be fall back or build upon a technology set proven out within a standards process.

Bloat or Riches?

Still, Java makes a developer spoiled. Often, and more so all the time, I find myself looking for a layer, wondering when said layer or its competitor becomes the reference implementation embodied in a JSR via the JCP. Why no generalized abstraction for blockchain yet? Why no generalized subsystem for communicating with AI/machine learning in the JSR yet? Unified security! It's here and developing, but whose implementation will win the coveted (I presume) position of "reference implementation"? I move and process a lot of data between systems. Wait a minute. Why not just make Camel a formal JSR? The Wildfly folks already offer it a "module" in their container.

Wait. Isn't that bloat? It's not bloat. It's an embarrassment of riches. That's a problem with which developers can live. Also, one does not have to use the EE container. If you need some layer/abstraction from EE, just stick with the SE base and pull in the layer you need. For instance, say you need to whip up a quick routine to pull in data from multiple sources (as above) and probably will never use this one-off utility again. Grab the JPA layer and be done. The extension and layers are just tools. Just because you have these tools does not compel a developer to use them. But, boy oh boy, when you are trying to get something done, it's good to know they are available. These tools? Just let the container

manage them for you. And, by the way, most containers load these layers/abstractions/tools in a "lazy" fashion, meaning they are on disk but not consuming CPU, memory or I/O until you specifically need that specific part of the standard. Disk space is cheap these days. Still, if it bugs you, try the Microprofile.

As for the above wishlist? Now that Java EE has stepped outside Oracle, I would anticipate that after the governance dust settles, these enterprise-level facilities will begin to accelerate just as the standard-level language and facilities have accelerated. A guess to be sure, but I would assert it's a pretty safe guess.

Current Status

The current status of things since the six minutes of terror this past summer?

A lot has been going on. Oracle found a home for the code base of the Java EE with the good folks over at the Eclipse Foundation. The Eclipse Foundation is also the home of a very nice IDE, an implementation of the JPA layer discussed above and lots of other neat projects. Many of the big guys are already on board with Eclipse and it being a steward for EE. For instance, Oracle is still on board and Red Hat and IBM are there too.

So far, they seem to have an open ear to concerns from the little guys like myself who are generally more focused on our keyboards and screens. When things seemed most uncertain, the community of developers formed the Java EE Guardians. This group first came to my attention when I read a post by Reza Rahman. Reza and the group, though concerned about the transition, really earned points for keeping cool and making a solid effort to be metered in dialogue with the Eclipse Foundation and Oracle. The Eclipse Foundation gets solid points for being active listeners and moving some non-engineering item like branding up on the agenda. Oracle, though it would have been nice to allow the donated project to retain the Java name and the package prefixes of javax, also gets some kudos for being direct with regard to its stance on these items. It could have strung the process out and been ambiguous, but it didn't. Hey, folks at Apache! Thank you for clearing the road for the use of the name Jakarta. Lots of great projects have evolved over at Apache under that umbrella.

Jakarta as the new brand makes it feel like a seamless transition, despite there having been many stakeholders and lots of moving parts.

In modern corporate speak: great EQ folks!

Java is alive and well. It is growing. OpenJDK seems not only to continue to ensure Java SE is viable, but it also leaves one with the impression that the Java core and ecosystem is accelerating.

Java EE is now Jakarta EE. A rose by another name? And, if OpenJDK is the model for Jakarta EE at Eclipse (under an umbrella project called EE4J), we should anticipate that the Jakarta EE standard soon will have its footing at its new home and will both accelerate and evolve as a standard too.

Rest easy. Let the fear, uncertainty and doubt on the future of both standard and enterprise Java fade away.

Welcome, Jakarta EE! ■

Dennis Gesker is a fan of Linux, open source, Java and all things tech. Dennis oversees technology efforts for Alamon, Inc., a telecommunications and utility services firm headquartered in Kalispell, Montana, where he lives with his lovely wife, Kelly. Dennis is easy to find on social media: LinkedIn (https://www.linkedin.com/in/gesker), WordPress (https://dennis.gesker.com), @Gesker on both Twitter and Facebook, and he is always happy to help other developers when he is able. We are a community after all!

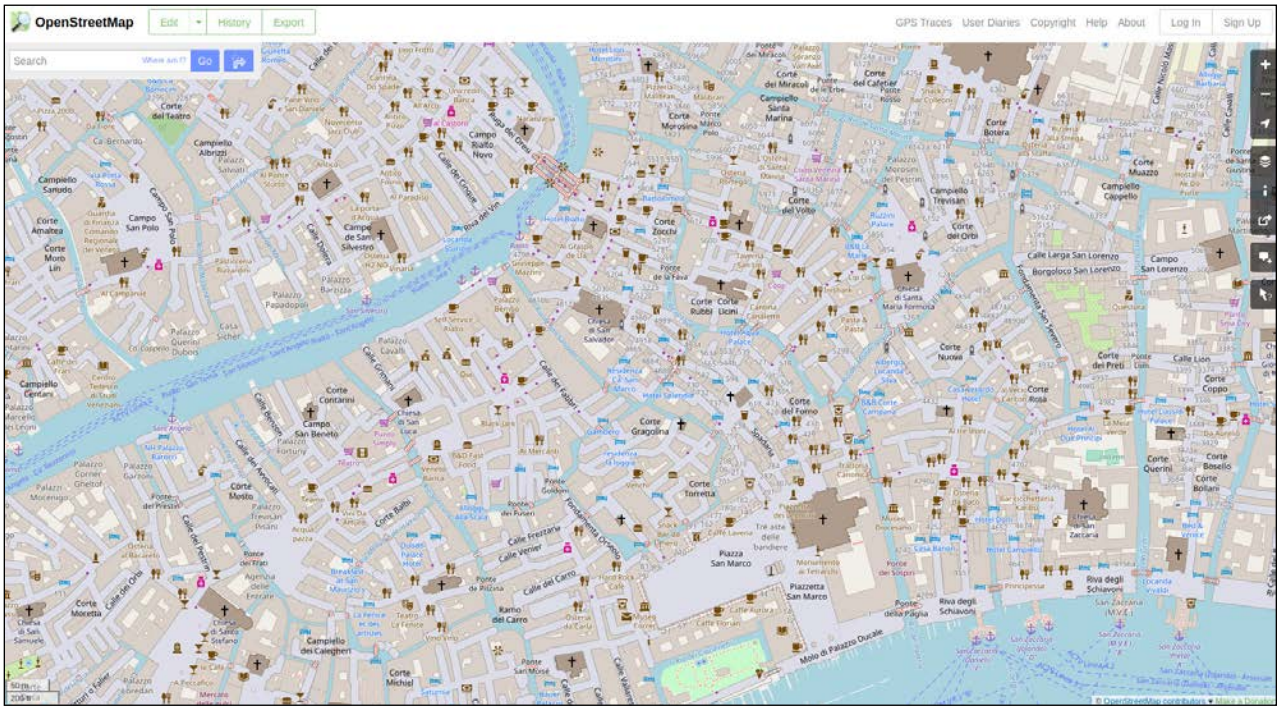
OpenStreetMap Should Be a Priority for the Open Source Community

Why open source needs an open geographic dataset.

By Glyn Moody

Glyn Moody has been writing about the internet since 1994, and about free software since 1995. In 1997, he wrote the first mainstream feature about GNU/Linux and free software, which appeared in *Wired*. In 2001, his book *Rebel Code: Linux And The Open Source Revolution* was published. Since then, he has written widely about free software and digital rights. He has a blog, and he is active on social media: @glynmoody on Twitter or identi.ca, and +glynmoody on Google+.

Open source has won. The fact that free software now dominates practically every sector of computing (with the main exception of the desktop) is proof of that. But there is something even more important than the victory of open source itself, and that is the wider success of the underlying approach it embodies. People often forget just how radical the idea of open, collaborative development seemed when it appeared in the 1990s. Although it is true that this philosophy was the norm in the very earliest days of the field, that culture was soon forgotten with the rapid rise of commercial computing, which swept everything before it in the pursuit of handsome profits. There, a premium was placed on maintaining trade secrets and of excluding competitors. But the appearance of GNU and Linux,



along with the other open software projects that followed, provided repeated proof that the older approach was better for reasons that are obvious upon reflection.

Open, collaborative development allows people to build on the work of others, instead of wastefully re-inventing the wheel, and it enables the best solutions to be chosen on technical, rather than commercial, grounds. The ability to work on areas of personal interest, rather than on those assigned by managers, encourages new talent to join projects in order to pursue their passions, while the non-discriminatory global reach of the open method means that the pool of contributors is much larger than for conventional approaches. However, none of those advantages is tied to software: they can be applied to many fields. And that is precisely what has happened in the last two decades, with the ideas underlying free software producing astonishing results elsewhere.

Arguably one of the most important knock-on effects of free software is the open nature of the World Wide Web. In his book, *Weaving the Web*, Tim Berners-Lee writes that he originally wanted to release his creation under the GPL and decided in 1993 to place it in the public domain only because key computing companies indicated that they wouldn't support the original idea. Also influenced by Richard Stallman's work is Paul Ginsburg, who set up arXiv.org in 1991 as an open repository of scientific preprints. This

idea of unrestricted sharing of academic knowledge later gave rise to the open access movement. Similarly, the GNU project's concepts and even name fed into Nupedia, the little-known precursor to Wikipedia. The latter's open collaborative approach has been so successful, it's hard to imagine modern life without this extraordinary resource.

Those are all major projects that are familiar to many people. But there's another application of the core ideas of openness and collaboration that is not as well known as it should be:

> OpenStreetMap is a world-wide collaborative project aiming at providing free map data, under an open license, to anyone who wants it. Volunteers all over the planet contribute their local knowledge and their time to build the best map ever.

OpenStreetMap began in 2004. It was inspired by Wikipedia, and it took off thanks to Linux, as its creator, Steve Case, explained in 2014:

> I gave a lot of talks. Linux user groups used to be very popular—people getting together on a Saturday afternoon to talk about Linux. They already knew half the story because they knew about open source and they knew about computers and data. So it wasn't too hard to explain what OpenStreetMap was doing.

Despite its low profile, OpenStreetMap is arguably one of the most important projects for the future of free software. The rise of mobile phones as the primary computing device for billions of people, especially in developing economies, lends a new importance to location and movement. Many internet services now offer additional features based on where users are, where they are going and their relative position to other members of social networks. Self-driving cars and drones are two rapidly evolving hardware areas where accurate geographical information is crucial. All of those things depend upon a map in critical ways, and they require large, detailed datasets. OpenStreetMap is the only truly global open alternative to better-known, and much better-funded geodata holdings, such as Google Maps.

The current dominance of the latter is a serious problem for free software—and freedom

itself. The data that lies behind Google Maps is proprietary. Thus, any open-source program that uses Google Maps or other commercial mapping services is effectively including proprietary elements in its code. For purists, that is unacceptable in itself. But even for those with a more pragmatic viewpoint, it means that open source is dependent on a company for data that can be restricted or withdrawn at any moment.

There is a more subtle problem with using a proprietary dataset. Free software is inherently about freedom because it allows users to do what they wish with a program. Proprietary code, by contrast, is under the control of the company that produced it, which means its users are subject to any constraints that are built in to the software: "Code is law", as Larry Lessig famously wrote. It is the same with mapping data and services. If the dataset and mapping software are proprietary, they come with the constraints and biases of the company that created them built-in, and they are impossible to circumvent. This might mean certain businesses are highlighted in an area because they have paid to be given preference. Services that help users find optimum routes also may be biased in socially harmful ways—for example, avoiding districts that the map provider deems "unsuitable".

The lack of a wider appreciation of the importance of OpenStreetMap to the future of open source is bad enough. But the situation seems to be even worse if a recent post by a long-time contributor to the project is correct. Serge Wroclawski worked on the OpenStreetMap project for around eight years, and he helped set up OpenStreetMap US, a nonprofit organization dedicated to OpenStreetMap in the United States. In 2014, he wrote a widely discussed article "Why the world needs OpenStreetMap", which explores many of the points mentioned above in greater detail, and with greater authority. It's well worth reading, as is his latest exploration of OpenStreetMap, ominously called "Why OpenStreetMap is in Serious Trouble". As he writes:

> While I still believe in the goals of OpenStreetMap, I feel the OpenStreetMap project is currently unable to fulfill that mission due to poor technical decisions, poor political decisions, and a general malaise in the project. I'm going to outline in this article what I think OpenStreetMap has gotten wrong. It's entirely possible that OSM will reform and address the impediments to its success—and I hope it does. We need a Free as in Freedom geographic dataset.

Leaving aside the political issues Wroclawski raises, one key point to emerge from his critical examination of OpenStreetMap's current status is that there are significant opportunities for the Open Source community to help take OpenStreetMap to the next level. Assuming Wroclawski's analysis is correct, there are plenty of ways programmers could help rectify some of the deficiencies of the project he identifies. In the process, they could tackle exciting and worthwhile new coding challenges. It often seems that many of the major free software projects have reached a slightly boring maturity, with all that this implies for offering sufficient incentives for people to join. The world of mapping, by contrast, has the potential to provide ambitious coders with the scope to make their mark in a new field of interest to billions of users.

So far, OpenStreetMap has spawned only a limited range of free software tools, and it badly needs many new ones if it is to match proprietary offerings. That's a relatively straightforward task. Much harder will be creating open versions of online services like Google's Waze, which describes itself as "the world's largest community-based traffic and navigation app". Doing so will require working with well-funded organizations—perhaps the Mozilla Foundation—or with open-source companies like Canonical, since the investment required will be considerable.

Although undoubtedly difficult, creating high-quality map-based services is a challenge that must be tackled by the Open Source community if it wants to remain relevant in a world dominated by mobile computing. The bad news is that at the moment, millions of people are happily sending crucial geodata to proprietary services like Waze, as well as providing free bug-fixes for Google Maps. Far better if they could be working with equal enthusiasm and enjoyment on open projects, since the resulting datasets would be freely available to all, not turned into corporate property. The good news is that OpenStreetMap provides exactly the right foundation for creating those open map-based services, which is why supporting it must become a priority for the Open Source world. ■

Send comments or feedback
via http://www.linuxjournal.com/contact
or email ljeditor@linuxjournal.com.